Distributed Algorithms and Optimization

Theoretical Foundations Module for Scalable Data Science and Distributed Machine Learning,
The fifth mandatory PhD course in the AI-Track of WASP Graduate School
`https://wasp-sweden.org/graduate-school/ai-graduate-school-courses/`
These lecture notes are an adaptation by Raazesh Sainudiin for WASP Graduate School
and based on a Stanford Course taught by Reza Zadeh and scribed by Robin Brown.

Reza Zadeh

November 18, 2020

Contents

1 Background

This section briefly covers or points to some of the background definitions and concepts required to appreciate the course. Most of this material would be normally encountered in a first course on *Algorithms and Data Structures* in any undergraduate program in Computer Science.

1.1 Computers and Algorithms

A *computer* is a machine that can be instructed to carry out sequences of arithmetic or logical operations to perform a specific task such as solving a particular problem. A computer *program* is a collection of instructions that can be executed by a computer. The method underlying a program to solve a particular problem that can be specified by a finite sequence of well-defined, unambiguous, computer-implementable instructions is known as an *algorithm*. Thus an algorithm can be viewed as a computational procedure that allows us to solve a *computational problem* specified by a given *input/output* relationship. For example, consider the *sorting problem* specified the following input/output relationship:
Input: A sequence of n numbers $(x_1, x_2, \ldots, x_n)$
Output: A reordering $(x_{(1)}, x_{(2)}, \ldots, x_{(n)})$ of the input sequence such that $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$

For example, given $(4, 3, 1, 2)$ as the input sequence or *instance* of the problem, the sorting algorithm should *return* as output the sorted sequence $(1, 2, 3, 4)$. An algorithm is said to be *correct* if it halts with the expected output for every input instance. For example, a sorting algorithm would be correct if it produced the sorted sequence for every one of the $n!$ input sequences of any given sequence of n numbers.

A correct algorithm is said to *solve* the given computational problem. Two algorithms can solve a problem, but one may solve it with fewer computer resources than the other. An algorithm that uses fewer resources than another is said to be more *efficient*. To appreciate algorithmic efficiency we need some understanding of the computer resources that are crucial to solving a computational problem.

This course assumes you have taken an intermediate to advanced undergraduate course in *Analysis of Algorithms* in a sequential single-machine setting. A timeless course is from the authors of the classical Book on the topic [1] or its modern versions [2]. It is impossible to learn the contents of this course without such basic undergraduate foundations in analysis of algorithms.

1.2 Computer architecture

Computers one encounters today include desktops, laptops, tablets, and smartphones. All such computers have the following common features. A *processor* is an electronic device capable of manipulating data as specified by a program. The processor requires *memory* to store the program

[1] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/index.htm

[2] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/index.htm

and data and *input/output (I/O) devices* such as keyboards, monitor, hard-drive, etc., along with support logic. These three main components are depicted in Fig. 1.

Nearly all modern processors are implemented on a single integrated circuit known as microprocessors or CPUs (Central Processing Units) including Intel, ARM, and Sun SPARC. The memory of the computer contains both the instructions that the processor will execute (to run the program) and the data it will manipulate. Thus, while instructions are read from memory, the data to be manipulated is both read from and written to memory by the processor. Most modern computers follow such a computer architecture known as the *von Neumann architecture* and are termed *control-flow computers* as they follow a step-by-step program that governs its operations.

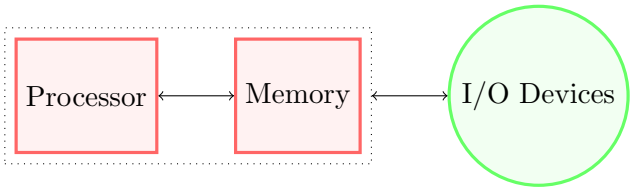


Figure 1: Simplified architecture of a computer

There are *four basic types of operations* that a processor can perform. The processor can (1) *write data* to memory or to an I/O device, (2) *read data* from memory or from an I/O device, (3) *read instructions* from memory, and (4) perform internal manipulation of data within the processor using the *Arithmetic Logic Unit (ALU)*. The processor has limited internal data storage known as its *registers* or *caches* and these are used to hold the data or operands that the processor is currently manipulating. The ALU performs the internal arithmetic and logical operations to manipulate the data in the processor. The instructions that are read from memory and executed by the processor not only control the data flow between the registers and the ALU but also the arithmetic operations performed by the ALU.

Clearly, the total number of operations performed by a processor in the course of implementing an algorithm is directly proportional to the time taken to produce the desired output from a given input. Thus, time taken by an algorithm is a clear measure of its efficiency.

We need to be aware of a other crucial aspects of computer architecture that can determine the efficiency of an algorithm. A *bus* is a data path in a computer, consisting of various parallel wires to which the processor, memory, and all input/output devices are connected. Thus, a bus determines how much and how fast data can move across the devices. For example, a processor can access data stored in caches much faster than it can from memory through a bus. In many systems, the processor makes no distinction between reading or writing data between memory and an I/O device such as a hard-disk. However there can be significant difference in the performance times. Reading data in memory can be orders of magnitude faster than reading data from external hard-disk. We will have to be mindful of such differences when designing efficient algorithms in a sequential, parallel as well as distributed computing models.

1 Introduction

1.1 Overview

The first set of lectures will be focused on analyzing the behaviour of algorithms in a typical personal computer today that is capable of *parallel* computing, and the second on a cluster of such personal computers in a typical public cloud today that is capable of *distributed* computing. We will start from first principles assuming minimal background covered briefly in §1 and work towards understanding notable algorithmic breakthroughs from the last several decades.

Parallel computing refers to computation with multiple processors and shared memory on a *single* machine. Although closely related, parallel and distribute computation both present unique challenges — chiefly, management of shared memory in the case of parallel computation and minimization of network communication overhead in the case of distributed computation. Understanding the models and challenges of parallel computation is foundational for understanding distributed computation. The course content reflects this by first covering various classical, numerical and graph algorithms in a parallel setting, and then covering the same topics in a distributed setting. The aim is to emphasize the unique challenges that each setting brings.

1.2 Introduction to Parallel Algorithms

Why focus on parallel algorithms? The regular CPU clock-rate[3], an indicator of the processor's speed, used to double every several years which meant our algorithms would run faster. This phenomena tapered off after CPUs reached around 3 Gigahertz; we reached fundamental limits on how much heat can be transferred away from the CPU. This is in contrast with memory and hard-disk space, which have not stopped getting bigger, yet; each year we can buy larger amounts of memory for the same price. CPUs are different, and so hardware manufacturers decided to have many cores instead of faster cores. Because computation has shifted from sequential to parallel, our algorithms have to change.[4]

Efficiency of Parallel Algorithms Even *notions of efficiency* have to adapt to the parallel. Typically, the efficiency of algorithms is assessed by the number of operations needed for it to halt with the correct output, under the assumption that this is a good proxy for wall-clock compute time. However, this is no longer an appropriate metric in the case of parallel algorithms. An efficient parallel algorithm may be inefficient when emulated on a single processor and vice-versa. Consequently, we will introduce a new measure of complexity for parallel algorithms, *work-depth*, and illustrate how algorithms can be designed to optimize work-depth.

Different types of hardware A machine made with an Intel instruction set will have dozens of processors or cores in it, each capable of complex operations; but notice it's only dozens, e.g.

[3] `https://en.wikipedia.org/wiki/Clock_rate`

[4] One might think we can take existing sequential algorithms and magically run a compiler to make it a parallel program. It turns out that writing such a compiler is actually an NP hard problem, something we will prove later on in this course.

16 or 32. Each processor in this system has its own local memory or cache and has access to a globally shared memory which contains the data and programs along with a table of processes (or sub-programs) awaiting execution. Each processor will obtain a process and associated data into its local memory and will run independently of the other processors. Inter-process communications can happen in the shared memory. These are known as Shared-Memory Multiple-Instruction Multiple-Data or `SM-MIMD` machines. An algorithm being executed on the `SM-MIMD` machine is usually divided into a set of smaller sub-problems, each executed on a processor. NVidia, on the other hand, pursues GPU computing, in which there are thousands of cores, each of which can only do basic floating point operations; at each time-step all of the cores must perform the same operation, with the caveat that they are feeding off of different data. These are known as Single-Instruction Multiple-Data or `SIMD` machines. There are many other variants as well.

1.3 Abstract Models of Computation

In order to analyze the efficiency of an algorithm it is usually formulated using an abstract model of computation or an *abstract machine*[5]. Such a model allows the designer of an algorithm to ignore many hardware details of the machine on which the algorithm will be run, while retaining enough details to characterize the efficiency of the algorithm, in terms of taking less time to complete successfully or requiring minimal resources. We will see two main models of computation: *sequential random-access machine (SRAM) model* and *parallel random-access machine (PRAM) model*.

1.3.1 Sequential Random-Access Machine (SRAM) Model

In sequential random-access machine (SRAM) model[6], the machine is abstracted to consist of a single processor, $\texttt{P}_1$, attached to a memory module, $\texttt{M}_1$, such that *any* position in memory can be accessed (read from or written to) by the processor *in constant time*. Furthermore, every processor operation (memory access, arithmetic and logical operations) by $\texttt{P}_1$ in an algorithm requires one time step and therefore all operations in an algorithm have to proceed sequentially step-by-step, such as when using `loop`s in C or Java to sum a list of integers. Thus the algorithm designer's goal in the SRAM model is to minimize performance metrics such as time taken and memory needed for the algorithm to successfully terminate. Such metrics are typically formulated as asymptotic functions of the problem input size. This is the model used in a first course on analysis of algorithms and we will assume familiarity with the SRAM model.

1.3.2 Parallel Random-Access Machine (PRAM) Model

The parallel random-access machine (PRAM) model[7] is a model of computation for a class of machines with two or more processors that are attached to a *shared memory*[8] module. Just like the

[5] `https://en.wikipedia.org/wiki/Abstract_machine`

[6] This is usually referred to as the random-access machine (RAM) model. We call it SRAM to emphasize the inherent Sequential aspect of the abstract machine and distinguish it from the PRAM model.

[7] `https://en.wikipedia.org/wiki/Parallel_random-access_machine`

[8] `https://en.wikipedia.org/wiki/Shared_memory`

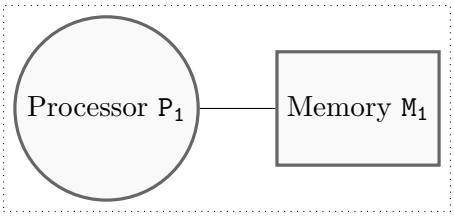


Figure 2: Sequential Random-Access Machine (SRAM) Model

SRAM model in a single processor sequential setting, the PRAM model allows designers of parallel algorithms to analyze their efficiency by ignoring many practical matters, including:

- the several layers of *cache* with different read-write speeds between each processor and its accessible memory
- the cost of other parallel programming overheads, such as, *synchronization*[9] of processes and data and *inter-process communication*[10].

A standard measure of efficiency will be the time taken by a parallel algorithm to terminate as a function of not only the problem input size (as before in the SRAM model), but also the number of processors. Thus, we can study how time taken can possibly decrease with more processors for a given input size.

Let us first note that there is still a clock which ticks for all processors at the same time. At each clock-tick, each processor can read or write to memory. Hence it is in fact possible for several processors to concurrently attempt to access a piece of memory (at the *exact* same time). This basic fact and the ensuing caveat on how exactly these multiple processors are allowed to instantaneously interact with the memory module(s) is captured by the PRAM model.

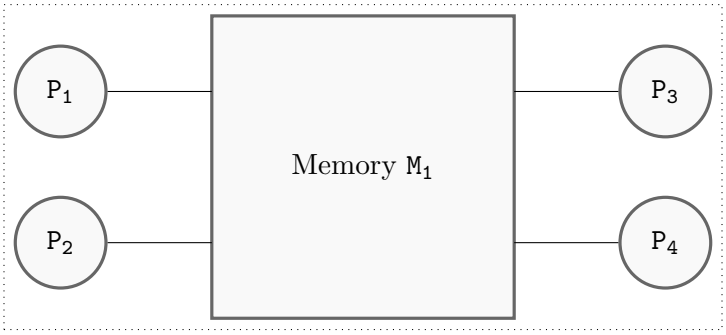


Figure 3: Parallel Random-Access Machine (PRAM) Model with shared memory

Caveat - Variants of PRAM Models In PRAM model we have multiple processors, each of which can perform any sequence of operations in the SRAM model: access a single large block of

[9] https://en.wikipedia.org/wiki/Synchronization_(computer_science)
[10] https://en.wikipedia.org/wiki/Inter-process_communication

the shared memory to read or write, perform arithmetical and logical operations. Crucially, now each processor can perform sequential operations independently of the others and at the same time, i.e., in parallel, with one caveat: when two processors want to access the same location in memory at the same time (whether it is read or write), we need to come up with a resolution. What type of resolution we come up with dictates what kind of parallel model we use. The different ways a PRAM model can be set up is the subset of the following combinations:

$$\{\text{Exclusive}, \text{Concurrent}\} \times \{\text{Read}, \text{Write}\}.$$

	Exclusive Read	Concurrent Read
Exclusive Write	Every memory cell can be read or written to by only one processor at a time	Multiple processors can read a memory cell but only one can write at a time
Concurrent Write	Never considered	Multiple processors can read and write

Resolving Concurrent Writes When dealing with concurrent writes, there needs to be a way to resolve when multiple processors attempt to write to the same memory cell at the same time. Here are the ways to deal with concurrent write:

- Undefined/Garbage: Machine could die or results could be undefined and impossible to interpret consistently and therefore considered as "garbage".
- Arbitrary: There is no predetermined rule on which processor gets write-priority. For example, if $\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_j$ all try to write to same location, we randomly select arbitrarily *exactly one* of them to give preference to.
- Priority: There is a predetermined rule by which processors gets to write, e.g., we give $\mathtt{P}_i$ priority to write over $\mathtt{P}_j$ if $i \leq j$.
- Combination: We write a combination of the values being written, e.g., max or logical `or` of bit-values written. Note that that unlike priority or arbitrary, the combination method relies on the *values* being written, not the processors doing the writing.

Of these combinations, the *most popular model* is the concurrent read and exclusive write (CREW) PRAM model. However, sometimes concurrent write is used. [11] When we use the PRAM model in the sequel, unless stated otherwise, we mean the CREW PRAM model.

Pros and Cons of Model Variants In the world of sequential algorithms, the model is fixed. However, in the parallel world, we have the freedom to decide which variant of the PRAM model to use. The benefit of this is that the algorithm designer may choose the model that is well-tailored to their application. The downside is that comparing algorithms across models is difficult. We'll see

[11] We note that some of these combinations don't make sense. For example, exclusive read and concurrent write.

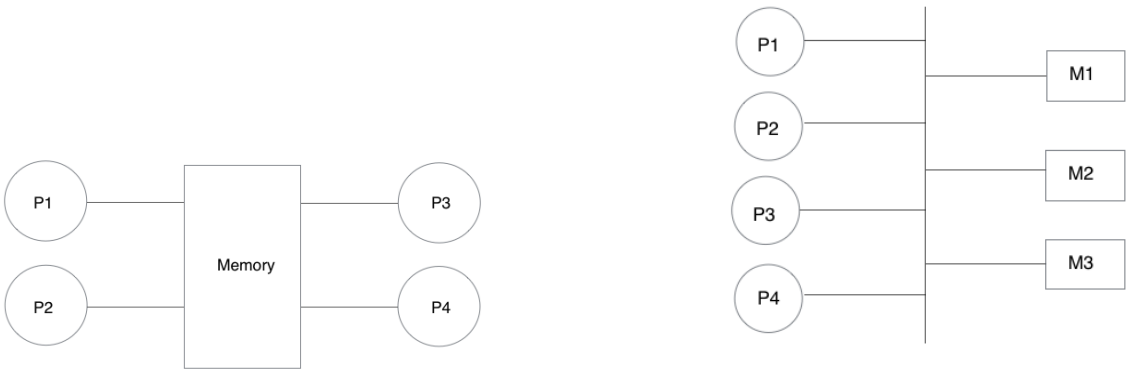
how a parallel algorithm's efficiency can be orders of magnitude different using different models for parallel computation. As an algorithm *designer*, you should advertise the model which you think your algorithm will be used on the most.

1.3.3 Other Multiprocessor Abstract Machine Models

PRAM model is a good model of computation when multiple processors can access a single shared memory module as in our smart phones or in multi-core processors in our laptops today. Below, we depict two examples of multiprocessor abstract machine models, besides the PRAM model, that are suitable for other types of parallel computer architectures. These models of computation should be used when designing efficient parallel algorithms on machines whose architecture violates the fundamental abstraction in the PRAM model that each processor can access any contiguous part of the shared memory unit in constant time.

On the right of Fig. 4, we have an abstract model of computation for a machine with multiple processors connected to multiple memory modules via a bus called the *modular memory machine model*. In this model each processor can access memory from *any* of the memory modules. On the left of Fig. 4, we have the *local memory machine model*. This model abstracts machines in which each processor has its own memory module, which cannot be directly accessed by other processors but instead may be accessed indirectly by communicating with the corresponding processor. In both of these models, a processor accesses a word of memory in one of the memory modules by sending a memory request through the bus or an appropriate interconnection network. The exact amount of time is not constant across all such memory accesses and depends on the communication network and memory access patterns across memory units (for example, local memory units may be more quickly accessible to the processor it is attached to). This is in stark contrast with the PRAM model where each processor can access any word of memory in a single step at constant time.

Figure 4: Examples of multiprocessor machine models that are distinct from the PRAM model (left); with local memory (not shown) and modular memory (right)



1.4 Work-Depth Model

We now move toward developing theory on top of these abstract models of computation. Instead of selecting one of the three multiprocessor models depending on the machine used for the implementation of our parallel algorithm — (i) PRAM model, (ii) local memory machine model or (iii) modular memory machine model — we can focus on the essence of parallel algorithms across all of them and define a more abstract multiprocessor model called the *work-depth model* which is devoid of superfluous machine-dependent details. The work-depth model guarantees that algorithms that are efficient on it can be translated to those that are also efficient in the three multiprocessor models up to constant factors. Next we use a natural way to bound the runtime of a parallel algorithm to motivate our definitions of *work* and *depth* before introducing the work-depth model that can represent parallel algorithms by directed acyclic graphs (DAGs).

1.4.1 Bounding runtime of parallel algorithms

In SRAM Models, we typically count the number of operations needed for an algorithm to terminate. This implicitly makes the assumption that the clock-time taken for the algorithm is proportional to the number of operations required, i.e.,

$$\text{time taken} \propto \text{ number of operations.}$$

This assumption is no longer valid for parallel algorithms. In a PRAM model, we have to wait for the last processor to finish all of its computation before we can declare that the entire computation is done. This is captured by the notion of *depth* of an algorithm as we will see. To capture this dependence between time taken to complete an algorithm and the number of processors used we define the following:

$$\begin{aligned} T_1 &:= \text{amount of (wall-clock) time algorithm takes on one processor} \\ T_p &:= \text{amount of (wall-clock) time algorithm takes on } p \text{ processors and} \\ T_\infty &:= \text{amount of (wall-clock) time algorithm takes on infinitely many processors.} \end{aligned} \tag{1}$$

1.4.2 Practical implications of work and depth

Defining work In general, work is the most important measure of the cost of an algorithm. The reasoning is as follows: the cost of a computer is proportional to the number of processors on that computer. The cost for purchasing time on a computer is proportional to the cost of the computer multiplied by the amount of time used. Hence the total cost of a computation is proportional to the number of processors in the computer multiplied by the amount of time required to complete all computations. This last product is the *work* of an algorithm.

Definition 1.1 (Work) *The work of an algorithm with p processors is defined to be the following processors-time product:*

$$W_p := p \times T_p \tag{2}$$

In words,

work := the number of processors used × amount of time required to complete all computations.

From the above definitions, observe that $W_1 = T_1$. The cost measured as work required with p processors will be typically slightly greater than that with one processor, i.e., $W_p > W_1$. However, with enough processors this extra cost can be justified as T_p can often be made to be less than T_1 and in the ideal case T_p can be T_1/p as explained next.

Fundamental lower bound T_p Note that in the best case, the total work required by the algorithm is evenly divided between the p processors; hence in this case the amount of time taken by each processor is evenly distributed as well. Fundamentally, T_p is lower bounded by

$$\frac{T_1}{p} \leq T_p \ , \tag{3}$$

where the equality gives the best case scenario.[12] This is the lower bound we are trying to achieve as algorithm designers. We would now like to determine a useful upper bound for T_p.

Relating Depth to PRAM with Infinite Processors In establishing theory, it's relevant to ask: what happens when we have an infinitude of processors? One might suspect the compute time for an algorithm would then be zero. But this is often not the case, because algorithms usually have an *inherently* sequential component to them. For example, suppose we represent an algorithm as a collection of computations, such that the output of one computation is used as the input to another, then the first must complete before the second can begin. Therefore, having infinitely many processors cannot avoid this inherent dependency of the second computation on the first.

1.4.3 Representing algorithms as a DAG

The above intuition can be made rigorous by representing the dependencies between inputs, computational operations and outputs in an algorithm using a directed acyclic graph (DAG) composed of nodes and arcs (directed edges attached to one or two nodes).

Constructing a DAG from an algorithm Each node represents a fundamental unit of computation assumed to have the same cost when performed on the operands arriving from its incoming arcs. Examples of such a computation could be one binary arithmetic or logical operation on two operands or their functional compositions seen as one operation performed on one or more operands. An arc *from* its *origination node u to* its *destination node v* exists if computation in u is required as an *input* to computation in v. The inputs to the algorithm are represented by a special

[12] To see this, assume toward contradiction that $T_p < T_1/p$, i.e. that $T_p \cdot p < T_1$. Note that T_p describes the time it takes for the entire algorithm to finish on p processors, i.e. the processor with the most work takes T_p time to finish. There are p processors. Hence if we were to perform all operations in serial it should require at most $T_p \cdot p$ time. But if $T_p \cdot p < T_1$, then we get a contradiction, for T_1 should represent the time it takes to complete the algorithm on a sequential machine.

set of arcs called *input arcs* and the outputs are represented by another special set of arcs called *output arcs*. The input and output arcs are only attached to one node. The input arcs do not have an originating node because memory access costs of input values are ignored in the work-depth model. Similarly, the output arcs do not have a destination node as the cost of reading the result is ignored. The origination node of an output arc is called an *output node* and the destination node of an input arc is called an *input node*. The output and input nodes are colored black and gray, respectively, in Fig.5. Note that more than one connected component can exist in the DAG if the underlying algorithm has different sets of input and output arcs for each component with no computational dependence between nodes across the two components as shown in Fig. 5.

Work and Depth of an Algorithm's DAG Suppose an algorithm is represented by the DAG $G := (V, E)$, with nodes in V and arcs in E. Given one processor, the *work* W_1 of an algorithm is exactly equal to the time taken T_1. Thus, when the algorithm is represented by G, W_1 and T_1 are given by $|V|$, the *size* of the DAG, i.e., the total number of nodes in the DAG. Given any number of processors, the *depth* of the DAG is given by the number of nodes in the longest directed path from an input arc to an output arc. Clearly, work captures the number of computational operations and depth captures the longest chain of dependencies. For a given node or arc in the Algorithm's DAG, let its *height* be the number of nodes along its longest directed path from any one of its input arcs. The set of all nodes with a given height can be said to be at the same *height-level* or *level* specified by their identical integer height. All nodes of the DAG in Fig. 5 are placed at their corresponding levels. When we proceed to complete the computation in the DAG, we typically start from the input nodes at level 1 and proceed to compute the nodes at the next level and so on until we reach each one of the output nodes. Thus, the depth of a DAG is the greatest of the heights of all output nodes. The depth of the DAG in Fig. 5 is 4 and its size is 21.

When DAG is a Tree A simple case occurs when there is only one *output node*, i.e., the origination node of the only output arc, and the DAG is a tree, as is the case of the second and third components of the DAG in Fig. 5. This output node is also called the *root node* of the tree and each *input node*, i.e., the destination node of one or more input arcs, is also called a *leaf node*. We will soon study a simple algorithm for adding numbers by representing it as a tree.

How an algorithm is executed on SRAM with a single processor In a sequential random-access machine (SRAM) model with a single processor, it's obvious what we must do to execute an algorithm. We look for the the input nodes of the DAG, since these depend on no prior computations but just a cost-free memory access from the input arcs. We may evaluate, i.e., perform the computation on, all of the input nodes and continue through each level of the DAG in ascending order sequentially until we reach each of the output nodes of the DAG.

How long does it take to execute the algorithm sequentially? Clearly, it takes time proportional to the number of nodes in the DAG (assuming each node represents a fundamental unit of computation which takes constant time). So, we see that T_1, which we defined to be the amount of time the algorithm takes on one processor to be *equal* to *work* W_1 which is given by the number of

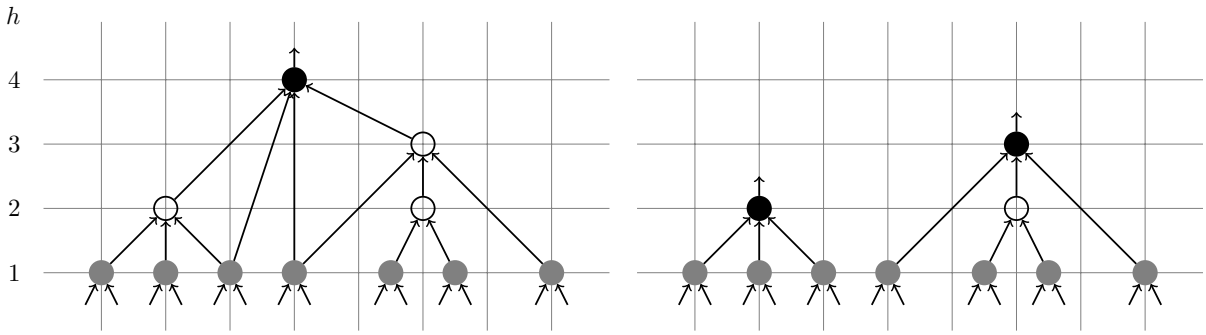


Figure 5: A Directed Acyclic Graph (DAG) representing an algorithm in which computations are done inside each node. There are three connected components, each with seven, three and four input nodes (colored gray) each with an incoming pairs of inputs arcs that represent the memory accesses of the input values. There are three output nodes (colored black) that perform the final computation for each connected component. Each output node has an outgoing output arc that returns the result of the computation. The three output arcs together give the Algorithm's final output. The height h of each node in the DAG is shown in the left.

nodes in the tree, as follows:

$$T_1 = W_1 = \text{size or number of nodes in DAG.}$$

How an algorithm is executed with an unlimited number of processors How much time would it take to execute the algorithm given an unlimited number of processors? Clearly, the depth of the DAG. At each level i, if there are m_i operations we may use m_i processors to compute all results in constant time. We may then pass on the results to the next level, use as many processors as required to compute all results in parallel in constant time again, and repeat for each level in the DAG. Note that we cannot do better than this.[13] So, with an infinitude of processors, the compute time is given by the depth of the DAG. We see that T_∞ which we defined to be the amount of time taken by the algorithm with infinitely many processors is equal to the *depth* of the DAG

$$T_\infty = \text{depth of the DAG.}$$

Realistically, the number of processors will be limited, so what's the point of T_∞? It is important as a fundamental lower-bound on the time an algorithm takes, and will be used in upper-bounding T_p.

[13]One feature that is considered universal across all machines is that at the fundamental hardware level, there is a discrete clock tick, and each operation executes exactly in sync with the clock. This feature, combined with the properties of our DAG, explain why the depth is a lower bound on compute time with infinite processors.

1.5 Brent's theorem

Theorem 1.1 *With T_1, T_p, T_∞ defined as above, if we assume optimal scheduling* [14]*, then*

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty. \tag{4}$$

Since T_1/p is optimal, we see that T_∞ allows us to assess how far off our algorithm performs relative to the best possible version of the parallel algorithm. T_∞ can be interpreted as *how parallel* an algorithm is.

This theorem is powerful because it tells us that if we can figure out how well our algorithm performs on an infinite number of processors, we can figure out how well it performs for any arbitrary number of processors. Ideally, we would go through each T_p to get a sense of how well our algorithm performs, but this is unrealistic.

Proof:(of Brent's Theorem) On level i of our DAG, there are m_i nodes. Then T_1 is equal to the size of the DAG,

$$T_1 = \sum_{i=1}^{d} m_i$$

where $T_\infty = d$, the depth of the DAG. For each level i of the DAG, the time taken by p processors is given as

$$T_p^{(i)} = \left\lceil \frac{m_i}{p} \right\rceil \leq \frac{m_i}{p} + 1.$$

This equality follows from the fact that there are m_i constant-time operations to be performed at m_i, and once all lower levels have been completed, these operations share no inter-dependencies. Thus, we may distribute or assign operations uniformly to our processors. The ceiling follows from the fact that if the number of processors is not divisible by p, we require exactly one wall-clock cycle where some but not all processors are used in parallel. Then,

$$T_p = \sum_{i=1}^{d} T_p^{(i)} \leq \sum_{i=1}^{d} \left(\frac{m_i}{p} + 1 \right) = \frac{T_1}{p} + d = \frac{T_1}{p} + T_\infty$$

■

So, if we *analyze* T_∞, and if we understand the sequential analysis of our algorithm which gives us T_1, we have useful bounds on how well our algorithm will perform on any arbitrary number of processors.

We lastly note that using more processors *can't worsen* run-time. That is, if $p_1 > p_2$, then $T_{p_1} \leq T_{p_2}$ (since we can always allow some processors to idle). Hence, the goal in the *work-depth model* is to design algorithms which work well on an infinitude of processors, since this minimizes T_∞, which in turn gets us closer to our desired optimal bound of T_1/p.[15]

[14] This is a non-trivial assumption because optimal scheduling is NP-hard. However, as we will see in next section, there is a constant approximation algorithm for optimal scheduling. This will allow us to reason about the asymptotic scaling of these algorithms despite not having an algorithm for optimal scheduling

[15] So although processors are a valuable resource, the work-depth model encourages us to imagine we have as many as we want at our disposal.

1.6 Parallel summation

How do we add up a bunch of integers? Sequentially, the summation operation on an array can be described by an algorithm of the form

Algorithm 1: Sequential Summation

```
1 s ← 0 for i ← 1, 2, ..., n do
2 |   s+ = a[i]
3 end
4 return s
```

For this summation algorithm $T_1 = n$. So the work of the algorithm is $O(n)$. What's T_2 on this algorithm? The only correct answer is $T_2 = n$ as well, since we haven't written the code in a way which is parallel. Further, realize that the depth of the algorithm is $O(n)$, since we have written it in a sequential order. In fact, $T_\infty = n$. So, if we increase the number of processors but keep the same algorithm, the time of algorithm does not change, i.e.

$$T_1 = T_2 = \cdots = T_\infty = n.$$

This is clearly not optimal. How can we redesign the algorithm?

$$\cdots \big(\big((a_1 + a_2) + a_3\big) + a_4\big) \cdots \ .$$

Instead of the Algorithm's DAG for the above summation being the left-branching binary tree with the same depth and size we can reduce the depth by first assigning each processor a pair of elements from our array, such that the union of the pairs is the array and there is no overlap. At the next level of our DAG, each of the summations from the leaf-nodes will be added by assigning each pair a processor in a similar manner. Note that if the array length is not even, we can simply pad it with a single zero. Realize that this parallel summation algorithm's DAG is the following balanced binary tree:

$$\cdots \big(\big((a_1 + a_2) + (a_3 + a_4)\big) + \big((a_5 + a_6) + (a_7 + a_8)\big)\big) \cdots \ . \tag{5}$$

When n is a power of 2 the DAG's T_1 given by its size is $n - 1$ and its T_∞ given by its depth is $\log_2 n$. Hence by Brent's theorem,

$$T_p \leq \frac{n}{p} + \log_2 n.$$

As $n \to \infty$, our algorithm does better since n/p dominates. As $p \to \infty$, our algorithm does not improve, since all that remains is the depth of $\log_2 n$.

1.7 A remark on associative binary operators

We remark that the parallel summation algorithm above with balanced binary tree as its DAG under the work-depth model also generalizes for any *associative binary operator*. For example, consider the `max` operator. It's associate and binary, and hence we may use a similar analysis as above to get the same efficiency.

2 Scalable algorithms, Scheduling, and a glance at All Prefix Sum

2.1 Types of scaling

Another fundamental quantity to understand is the idea of how much speed-up we can hope to achieve given more processors.

Definition 2.1 (SpeedUp) *Let $T_{1,n}$ denote the run-time on one processor given an input of size n. Suppose we have p processors. We define the speed up of a parallel algorithm as*

$$\text{SpeedUp}(p,n) := \frac{T_{1,n}}{T_{p,n}} \ .$$

There are three different types of scalability:(1) *strongly scalable*, (2) *weakly scalable*, and (3) *embarrassingly parallel*.

Definition 2.2 (Strongly Scalable) *If* $\text{SpeedUp}(p,n) = \Theta(p)$, *we say that the algorithm is strongly scalable.*

Claim 2.1 *The parallel sum of n numbers on p processors as per algorithm in* (5)*, is strongly scalable.*

Proof: To see this, recall that size is $T_1 = n$, and depth is $T_\infty = \log_2 n$, so $\text{SpeedUp}(p,n) = \frac{n}{\frac{n}{p}+\log_2 n} = \Theta(p)$. Specifically, both n and p are assumed to be going to infinity,but p grows much slower than does n. Hence

$$\frac{n}{\frac{n}{p}+\log_2 n} \geq \frac{n}{\frac{n}{p}+\frac{n}{p}} \geq \frac{p}{2} \qquad \text{and} \qquad \frac{n}{\frac{n}{p}+\log_2 n} \leq \frac{n}{\frac{n}{p}} = p.$$

■

Definition 2.3 (Weakly Scalable) *If* $\text{SpeedUp}(p,np) = \frac{T_{1,n}}{T_{p,np}} = \Omega(1)$*, then our algorithm is weakly scalable.*

This metric characterizes the case where, for each processor we add, we add more data as well. This is a useful metric in practice, because oftentimes the only time we can afford to add more processors or machines is when we are burdened with more data than our infrastructure can handle.

Definition 2.4 (Embarrassingly Parallel) *If the DAG representing an algorithm has a depth of* 1*, then the algorithm is said to be embarrassingly parallel.*

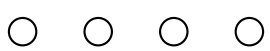


Figure 6: An Embarrassingly Parallel DAG (input and output arcs are not shown)

Thus in an embarrassingly parallel DAG, there is no dependency between the nodes representing computations or operations. It is scalable in the most trivial sense, e.g. doing a computation with the same cost many times over different inputs.

We note here that we have used Brent's theorem to derive the scaling bounds for the parallel sum algorithm. In the previous section, we alluded to the fact that Brent's theorem assumes optimal scheduling, which is NP-hard. Fortunately, the existence of a polynomial time constant approximation algorithm for optimal scheduling implies that these bounds still hold.

2.2 Scheduling

In addition to building algorithms with low depth, clever scheduling is just as important to parallelism. Given a DAG of computations, at any level in the DAG there are a certain number of computations which can be required to execute (at the same time). The number of computations is not necessarily equal to the number of processors you have available to you, so you need to decide how to assign computations to processors — this is what is referred to as scheduling [16].

Ideally, you wish for all your processors to be busy, however, depending on how jobs are assigned to processors, you might end up with processors that are idle and not working. Depending on the size and dependencies of jobs to be scheduled, it may not be possible for all processors to be busy all the time. This then turns into an optimization problem where we try to schedule jobs in a way that minimizes the idle time of processors. This problem turns out to be NP hard [17].

It is the scheduler's task to schedule things in tandem in such a way to minimize the idle time of processors. We could do this greedily, i.e., as soon as there is any computation to be done, we assign it to a processor. Or, we can look ahead in our DAG to see if we can plan more efficiently.

Spark, the system for distributed computing we will see in the sequel, has a scheduler. Every distributed computing set up has a scheduler. Your operating system and phone have schedulers. Every computer has processes, and every computer runs in parallel. Your computer might have fifty Chrome tabs open and must decide which one to give priority to in order to optimize performance of your machine.

2.2.1 Problem definition

An important problem in any parallel or distributed computing setting is figuring out how to schedule jobs optimally. I.e. a scheduler must be able to assign sequential computation to processors or machines in order to minimize the total time necessary to process all jobs.

Notation We assume that the processors are identical (i.e. each job takes the same amount of time to run on any of the processors). 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

[16] `https://en.wikipedia.org/wiki/Scheduling_(computing)`

[17] `https://en.wikipedia.org/wiki/NP-hardness`

for processor i is therefore, $L_i = \sum_{k \in S_i} J_k$. The goal is to minimize the *makespan* defined as $L_{max} = \max_{i \in \{1,\ldots,p\}} L_i$.

2.2.2 The simple (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 processor 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*.

Algorithm 2: Simple scheduler

```
for each job that comes in (streaming) do
    Assign job to lowest burdened machine
end
```

Other variants of scheduling We note there are many other variants of scheduling. Jobs can have *dependencies*, i.e. one job must finish before another job can start. Here, the problem is pre-specified by a computational DAG that is known before the time of scheduling. Another variant is that scheduling must happen *online*, i.e. jobs come at you in an order where you cannot look into the future. As jobs come in, you have to schedule it, and you cannot go back and change the schedule. For a comprehensive treatment of variants of scheduling, see Handbook of Scheduling.[18]

2.2.3 Optimality of the greedy approach

In either of the above cases, where jobs have dependencies or must be scheduled online, the problem is NP hard. So, we use approximation algorithms. We claim that the simple (greedy, and online) algorithm actually has an *approximation ratio* of 2. In other words, the algorithm is in the worst-case 2 times worse than the optimal, which is fairly good. 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 an approximation ratio of 2.

Proof: We first want to get a handle (lower bound) on OPT. We know that the optimal makespan must be at least the sum of the processing times for the jobs divided amongst the p processors,

[18] To give an idea of another variant, consider the case of distributed computing, where each machine houses a set of local data, and shuffling data across the network is a bottleneck. We may consider scheduling jobs to machines such that no data are shuffled. We'll consider this more in the latter part of the course. This is called *locality sensitive scheduling*

i.e.[19]

$$\text{OPT} \geq \frac{1}{p} \sum_{i=1}^{n} J_i. \tag{6}$$

A second lower bound is that OPT is at least as large as the time of the longest job, [20]

$$\text{OPT} \geq \max_i J_i. \tag{7}$$

Now consider running the greedy algorithm and identifying the processor responsible for the makespan of the greedy algorithm (i.e. k = $\arg\max_i L_i$). 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 $L_{\max} - J_t$, i.e. $L_{\max} - J_t \leq L'_i$ for all i. Hence, summing this inequality over all i,

$$p(L_{\max} - J_t) \leq \sum_{i=1}^{p} L'_i \leq \sum_{i=1}^{p} L_i = \sum_{i=1}^{n} J_i \tag{8}$$

In the second inequality, we assert that although J_t the last job placed on the bottleneck processor, there may still be other jobs yet to be assigned, hence we have that the sum of total work placed on each machine cannot decrease after assigning all jobs. 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:

$$L_{\max} \leq \frac{1}{p} \sum_{i=1}^{n} J_i + J_t \tag{9}$$

Now, note that our greedy algorithm actually has makespan exactly equal to $L_{\max}$, by definition. Using equations 6 and 7 along with the fact that $J_t \leq \max_i J_i$, we get that our greedy approximation algorithm has makespan

$$\text{APX} = L_{\max} \leq \text{OPT} + \text{OPT} = 2 \times \text{OPT}. \tag{10}$$

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

[19] To see this, assume toward contradiction that OPT is able to schedule jobs such that OPT $< \frac{1}{p} \sum_{i=1}^{n} J_i$. Suppose that instead of OPT assigning jobs to p processors in parallel, we assigned all the work to one processor sequentially. Then of course the total time required given by $p \cdot \text{OPT} < \sum_{i=1}^{n} J_i$ but this is a contradiction, since $\sum_{i=1}^{n} J_i$ exactly represents the amount of work required to process all n jobs on a single processor.

[20] The reason for this is simple: the longest job must be scheduled at some point to be run sequentially on one processor, at which point it will require $\max_i J_i$ time to compute. There may be other processors which bottleneck our makespan, but we know that OPT must take at least as long as any job, and in particular this holds for the largest job.

What if we could see the future? We note that if we first sort the jobs in descending order and assign larger jobs first, we can naively get a $3/2$ approximation. The intuition is that if we first schedule large jobs, we can use the smaller jobs to "fill in the gaps" remaining, i.e. to balance all loads. If we use the same algorithm with a tighter analysis, we get a $4/3$ approximation. We'll see later in the course how Spark uses lazy evaluation for exactly this reason: by faking computation until the user takes an *action*, Spark can sort jobs and to obtain a more efficient scheduler.

What's realistic? It may seem that our above assumption, to be able to look into the future and know which jobs are going to be scheduled, is quite unreasonable. In reality, we don't even know how long each job will take. However, we often have a pretty good idea (based on historical data or expectations) how long a particular job will take to run. And further, we may have statistics or expectations on how many jobs of a particular type are going to come in, hence, it may not be such an unrealistic scenario to know (within a certain tolerance) the expected amount of time each job will take as well as what jobs might be in the pipeline.

References

[1] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[2] G. Blelloch and B. Maggs. *Parallel Algorithms.* Carnegie Mellon University.

References

[1] Ola Svensson *Approximation Algorithms.* EPFL, January 21, 2013.

[2] Joseph Y-T. Leung. *Handbook of Scheduling.* CRC Press, 2004.

[3] G. Blelloch and B. Maggs. *Parallel Algorithms.* Carnegie Mellon University.

3 All Prefix Sum

Given a list of integers, we want to find the sum of all prefixes of the list, i.e. the running sum. We are given an input array A of size n elements long. Our output is of size $n+1$ elements long, and its first entry is *always* zero.

As an example, suppose $A = [3, 5, 3, 1, 6]$, then `AllPrefixSum`$(A) = [0, 3, 8, 11, 12, 18]$.

This feels like an inherently sequential task. The obvious way to do this with a single processor is to have a running sum and write intermediary sums as we iterate through the array in linear time. However, this does not parallelize at all. How can we parallelize this, so that it has low-depth?

3.1 Algorithm Design

In sequential world, this is trivial. How can we parallelize this problem? We need a mix of divide and conquer and our first parallel summation algorithm. We design the following algorithm.

Algorithm 3: Prefix Sum

Input: All prefix sum for an array A

if *size of A is 1* **then**
  return *only element of A*
end
Let A' be the sum of adjacent pairs
Compute $R' =$ AllPrefixSum(A') `// Note: R' has every other element of R`
Fill in missing entries of R' using another $\frac{n}{2}$ processors

A note on the size of our (sub)-problems The general idea is that we first take the sums of adjacent pairs of A. So the size of A' is exactly half the size of A. Note that if the size of A not a power of 2, we simply pad it with zeros. Notice that R' has every other element of R, our desired output.

Pairing gets a running sum for even parity indices Specifically, every element in R' corresponds to an element with an index of even parity in A. It's as though as we did a running sum, but we only reported the running sum every two iterates. That is, we have an array A and R

$$A = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_n \end{bmatrix}$$
$$R' = \begin{bmatrix} r_2 & r_4 & \dots & r_{n/2} \end{bmatrix}$$

That is, $r_2 = a_1 + a_2$, and $r_4 = a_1 + a_2 + a_3 + a_4$, and in general $r_k = \sum_{i=1}^{k} a_i$.

Filling in the odd-indices To compute the running sum for elements whose index is of odd parity in A, i.e. set

$$r_i = r_{i-1} + a_i$$

for $i = 1, 3, 5, \ldots$, where we by convention let $r_0 = 0$.

3.2 Algorithm Analysis

Our base case requires constant work and depth. Let's consider work and depth for each remaining step of the algorithm.

Pairing entries Notice that step 4, where we let A' be the sum of adjacent pairs, we must perform $n/2$ summations, hence work is $O(n)$. Realize that we may assign each processor a pair of numbers and perform the summations in parallel. Hence depth is $O(1)$.

Recursive call Step 5 is our recursive call, which is fed an input of half the size of A.

Filling in missing entries Notice that step 6, filling in missing entries, we can assign each of the $n/2$ missing entries of R to a processor and compute its corresponding value in constant time. Hence step 6 has work is $n/2$, i.e. $O(n)$, and depth $O(1)$.

Total work and depth Now let's consider the work and depth for the entire algorithm. This is where recurrences come into play. Let $T_1 = W(n)$, and $T_\infty = D(n)$,

$$W(n) = W(n/2) + O(n) \implies W(n) = O(n),$$
$$D(n) = D(n/2) + O(1) \implies D(n) = O(\log(n)).$$

The expression for work follows since we make exactly one recursive call of exactly half the size, and in outside our recursive call we perform $O(n)$ work. By the Master Theorem, $W(n) = O(n)$.[21]

With regard to depth, again realize that we make a recursive call on input size $n/2$, and outside the recursive call we only require constant depth. Again by the Master Theorem, we see that $D(n) = O(\log n)$. For prefix-sum, this is pretty much the best we can hope for. We emphasize that recursion was critical for the parallelization of this algorithm.

A note on recursive algorithms and parallelization

We conclude with the remark that although recursive algorithms are amenable to parallelization, the algorithm designer must do some analytical work to make algorithms efficient in a parallel setting. Often times, we the combine step of a divide-and-conquer algorithm is sequential in nature, and can be the bottleneck of our analysis. To get around this, we must think carefully. We'll see more on this when we talk about MergeSort next lecture.

4 Mergesort

Merge-sort is a very simple routine. It was fully parallelized in 1988 by Cole.[5] The algorithm itself has been known for several decades longer.

[21] Again, unrolling our recurrence we yield a geometric series scaled by n, hence work is $O(n)$.

4.1 The mergesort algorithm

Algorithm 4: Merge Sort

Input : Array A with n elements
Output: Sorted A

```
1  n ← |A|
2  if n is 1 then
3  |  return A
4  end
5  else
   |  // (IN PARALLEL, DO)
6  |  L ← MERGESORT(A[0,...,n/2))        // Indices 0, 1, ..., n/2 − 1
7  |  R ← MERGESORT(A[n/2,...,n))        // Indices n/2, n/2 + 1, ..., n − 1
8  |  return MERGE(L,R)
9  end
```

4.2 Subroutine: merge

It's critical to note how the `merge` sub-routine works, since this is important to our algorithms work and depth. We can think of the process as simply "zipping" together two sorted arrays.

Algorithm 5: Merge

Input : Two sorted arrays A, B each of length n
Output: Merged array C, consisting of elements of A and B in sorted order

```
1  a ← pointer to head of array A (i.e. pointer to smallest element in A)
2  b ← pointer to head of array B (i.e. pointer to smallest element in B)
3  while a, b are not null do
4  |  Compare the value of the element at a with the value of the element at b
5  |  if value(a) < value(b) then
6  |  |  add value of a to output C
7  |  |  increment pointer a to next element in A
8  |  end
9  |  else
10 |  |  add value of b to output C
11 |  |  increment pointer b to next element in B
12 |  end
13 end
14 if elements remaining in either a or (exclusive) b then
15 |  Append these sorted elements to our sorted output C
16 end
17 return C
```

Since we iterate over each of the elements exactly one time, and each time we make a constant time comparison, we require $\Theta(n)$ operations. Hence the `merge` routine on a single machine takes $O(n)$ work.

4.3 Naive parallelization

Suppose we parallelize the algorithm via the obvious divide-and-conquer approach, i.e. by delegating the recursive calls to individual processors. The work done is then

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ &= O(n \log n) \end{aligned}$$

by case 2 of the Master Theorem.

As you'll recall from earlier algorithms classes, the canonical implementation of the `merge` routine involves simultaneously iterating over L and R: starting at the first index of each, we merge them by placing the smaller of the currently pointed-to elements of L and R at the back of a new list and advance the pointer in the list that the just-placed element belonged to, and continue until we reach beyond the end of one list. Crucially, `merge` has depth $O(n)$. The depth is then

$$\begin{aligned} D(n) &= D(n/2) + O(n) \\ &= O(n) \end{aligned}$$

again by the Master Theorem.

Using Brent's theorem, we have that

$$T_p \leq O(n \log n)/p + O(n)$$

Therefore $W(n) = O(n \log n)$ and $D(n) = O(n)$.

The bottleneck is in sequential merge subroutine Note that the bottleneck lies in `merge`, which takes $O(n)$ time. That is, even though we have an infinitude of processors, the time it takes to merge two sorted arrays of size $n/2$ on the first call to `mergeSort` dominates the time it takes to complete the recursive calls.

4.4 Improved parallelization

How do we merge L and R in parallel? The `merge` routine we have used is written in a way that is inherently sequential; it is not immediately obvious how to interleave the elements of L and R together even with an infinitude of processors.

Using binary search to find the rank of an element Let us call the output of our algorithm M. For an element x in R, let us define $\text{rank}_M(x)$ to be the index of element x in output M. For any such element $x \in R$, we know how many elements (say a) in R come before x since we have sorted R. But we don't immediately know the rank of an element x in M.

If we know how many elements (say b) in L are less than x, then we know we should place x in the $(a+b)^{th}$ position in the merged array M. It remains to find b. We can find b by performing a binary search over L. We perform the symmetric procedure for each $l \in L$ (i.e. we find how many elements in R are less than it), so for a call to `merge` on an input of size n, we perform n binary searches, each of which takes $O(\log n/2) = O(\log n)$ time.

$$\underset{M}{\text{rank}}(x) = \underset{L}{\text{rank}}(x) + \underset{R}{\text{rank}}(x)$$

4.5 Parallel merge

Algorithm 6: Parallel Merge

Input : Two sorted arrays A, B each of length n
Output: Merged array C, consisting of elements of A and B in sorted order

```
for each a ∈ A do
    Do a binary search to find where a would be added into B,
    The final rank of a given by rank_M(a) = rank_A(a) + rank_B(a).
end
```

Analysis of parallel merge To find the rank of an element $x \in A$ in another sorted B requires $O(\log n)$ work using a sequential processor. Notice, however, that each of the n iterations of the for loop in our algorithm is independent of the previous, hence our binary searches may be performed in parallel. That is, we can use n processors and assign each a single element from A. Each processor then performs a binary search with $O(\log n)$ work. Hence in total, this parallel merge routine requires $O(n \log n)$ work and $O(\log n)$ depth.

Hence when we use `parallelMerge` in our `mergeSort` algorithm, we realize the following work and depth, by the master theorem:

$$\begin{aligned} W(n) = 2W(n/2) + O(n \log n) &\implies W(n) = O(n \log^2 n), \\ D(n) = D(n/2) + \log n &\implies D(n) = O(\log^2 n). \end{aligned}$$

By Brent's Theorem, we get

$$T_p \leq O(n \log^2 n)/p + O(\log^2 n)$$

so for large p we significantly outperform the naive implementation! The best known implementation (work $O(n \log n)$, depth $O(\log n)$) was found by Richard Cole[5].

4.6 Motivating Cole's mergesort

We notice that we use many binary searches in our recently defined parallel merge routine. Can we do better? Yes. Let L_m denote the median index of array L. We then find the corresponding index in R using binary search with logarithmic work. We then observe that all of the elements in L at or below L_m and all of the elements in R at $\text{rank}_R(\texttt{value}(L_m))$ are at most the value of L's median element. Hence if we were to recursively merge-sort the first L_m elements in L along with the first $\text{rank}_R(\texttt{value}(L_m))$ elements in R, and correspondingly for the upper parts of L and R, we may simply append the results together to maintain sorted order. This leads us to Richard Cole (1988).[5] He works out all the intricate details in this approach nicely to achieve

$$W(n) = O(n \log n)$$
$$D(n) = O(\log n)$$

5 Divide and Conquer Recipe, Parallel Selection

Today we will continue covering divide and conquer algorithms. We will generalize divide and conquer algorithms and write down a general recipe for it. What's nice about these algorithms is that they are *timeless*; regardless of whether Spark or any other distributed platform ends up winning out in the next decade, these algorithms always provide a theoretical foundation for which we can build on. It's well worth our understanding.

- Parallel merge sort
- General recipe for divide and conquer algorithms
- Parallel selection
- Parallel quick sort (introduction only)

Parallel selection involves scanning an array for the kth largest element in linear time. We then take the core idea used in that algorithm and apply it to quick-sort.

5.1 Parallel Merge Sort

Recall the merge sort from the prior lecture. This algorithm sorts a list recursively by dividing the list into smaller pieces, sorting the smaller pieces during reassembly of the list. The algorithm is as follows:

Algorithm 7: MergeSort(A)

Input : Array A of length n
Output: Sorted A

```
1 if n is 1 then
2 |   return A
3 end
4 else
5 |   L ← mergeSort(A[0, ..., n/2))
6 |   R ← mergeSort(A[n/2, ..., n])
7 |   return Merge(L, R)
8 end
```

Last lecture, we described one way where we can take our traditional `merge` operation and translate it into a `parallelMerge` routine with work $O(n \log n)$ and depth $O(\log n)$. Recall that to do this, we assign each of n processors a single element in one sorted array A and perform a binary search on the other array B for where the index would belong; the rank of the element in the output array is simply the sum $\text{rank}_A(x) + \text{rank}_B(x)$. Unfortunately, this leads our MergeSort routine to require work $W(n) = O(n \log^2 n)$, albeit at a better depth of $D(n) = O(\log^2 n)$. This is

bothersome since by Brent's theorem, we still have T_1 showing up in the upper bound, hence we won't want to increase work for our algorithms in general.

We also started to describe a way in which if we have two sorted arrays in memory, and the largest element of one array is less than the smallest element of the other, then we may simply append the arrays together to yield a sorted output. This led to Richard Cole's 1988 paper, which achieves $W(n) = O(n \log n)$ and $D(n) = O(\log n)$.[5] The way Cole was able to do this was by modifying his routine to return *more* information than just a sort alone. So, some of the work of the `Merge` routine was being performed in the `mergeSort` routine itself.

5.2 General Divide and Conquer Technique

Many divide and conquer algorithms are actually quite formulaic. We go over the general method.

1. Come up with recursive formulation of the problem.
2. Count the following:
   - Let a denote the number of recursive calls the function makes.
   - Let b denote the integer such that the recursive calls accept input of size n/b.
3. Determine the amount of work $T_1 = w$ and depth $T_\infty = t$ is required by procedures *outside of* any recursive calls; i.e. what is required of our *combine* step?
4. Write down the recurrence relations for the whole algorithm, which may be solved using the Master Theorem.[22]

$$T_1 = W(n) = aW\left(\frac{n}{b}\right) + w$$
$$T_\infty = D(n) = D\left(\frac{n}{b}\right) + t$$

   Part of the beauty in the divide and conquer method is that with respect to depth, it doesn't matter how many recursive calls we end up making. Since the recursive calls are independent, when we add them to our computational DAG, we do not incur more than one unit of depth.

Generally speaking, coming up with the recursive formulation takes the most creativity. The rest of the steps are more formulaic. It's not trivial to realize that to sort a bunch of numbers, you need to sort the left hand side and the right hand side and then merge the results back together. Many parallel algorithms, such as finding the convex hull of a bunch of points in 2D, are best solved using divide and conquer. Realizing that you even should be using divide and conquer takes a little bit of creativity.

As far as this class is concerned the hardest part is coming up with a combine step which is "cheap", i.e. does not require much work/depth. We're going to practice this creativity as we think about the next problem: parallel quick selection.

[22] It's also worth mentioning that unrolling recurrences by hand is not that difficult. If you ever forget which case of the Master Theorem to use, you should be able to figure it out from scratch without much trouble.

5.3 Parallel Quick Selection

Suppose we have a list of unsorted integers, which we know nothing about. We wish to find the k^{th} largest element of the integers. We can use *quick selection* to accomplish this task in linear time without having to sort the entire list.[3] Manuel Blum proved this result in 1971 and was awarded a Turing Award for it. Incidentally, he was actually trying to prove that you need to sort to be able to do arbitrary selection.

Quick selection is a randomized algorithm, so we will consider its expected run-time. We will solve this algorithm using Divide and Conquer.

We assume that our input array A has unique elements.[23]

Idea We summarize the algorithm in words. From the input list, A, pick a value at random called a pivot, p. For each item in A, put item into one of two sublists L and R such that:

$$x \in L \iff x < p$$
$$y \in R \iff y > p$$

We do not require L and R to be sorted. We can do this in linear time since it only requires passing through each element in A and performing a constant time check whether the element is greater or less than our pivot p.

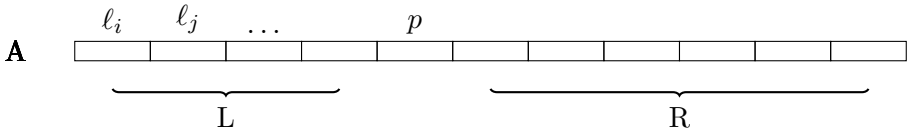


Figure 7: After re-arranging, all elements in L (such as ℓ_i, ℓ_j) are less than the pivot p. All elements in R are greater than the pivot p.

Crucial Realization Notice that the rank of p is *exactly* $|L|$. To find the k^{th} largest element in A, call it z, note the following:

$$\text{If } |L| < k, \text{then } z \notin L. \text{ We discard the values in L.}$$
$$\text{If } |L| > k, \text{then } z \notin R. \text{ We discard the values in R.}$$

We elaborate on why if $|L| < k$, then $z \notin L$. This is because for any element $l \in L$, its sorted order must be less than p's order. But p's order is equal to $|L| + 1$, so $z \in L$ would imply $k < |L|$, which is a contradiction. By similar logic we can argue if $|L| > k$, then $z \notin R$.

We then recursively perform the selection algorithm on the not discarded list, which reduces our problem size. Specifically, we prove that with probability 1/2, we reduce our problem size by at least 3/4. That will then be enough to guarantee expected linear time work.

[23] If the elements are not unique, then we can simply omit the duplicate elements from the recursive calls.

Algorithm 8: Select(A, k)

Input : Array A with n entries, integer $k \leq n$
Output: Value of the kth largest element in A

```
p ← a value selected uniformly at random from A     // This is our pivot.
L ← elements in A which are less than pivot p       // Requires Θ(n) work
R ← elements in A which are larger than pivot p
if |L| is k − 1 then
    return p                          // The pivot itself is the kth largest
       element
end
if |L| > k then
    return Select(L, k)               // kth largest element lives in L
end
else
    return Select(R, k-|L|-1)         // kth largest element lives in R
end
```

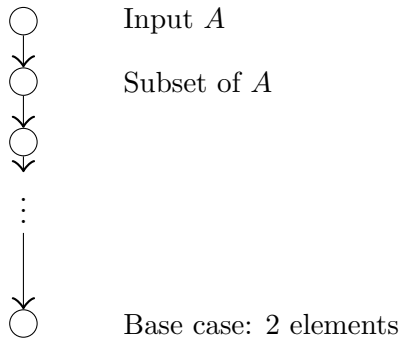


Figure 8: Observe the computational DAG for our `QuickSelect` algorithm, which is serial in nature.

Intuition for Analysis Since `Select` makes a chain of recursive calls, let us group multiple calls together into one "phase" of the algorithm. The sum of the work done by all calls is then equal to the sum of the work done by all phases of the algorithm. Concretely, let us define one "phase" of the algorithm to be when the algorithm decreases the size of the input array to 3/4 of the original size or less.

Why 3/4? Notice that if we can shrink the size of the array by a constant factor each iteration while only performing linear work per iteration, then using the Master Theorem we know that total work done would be linear. Further, picking any pivot in the "middle" half of the array means that after re-arranging elements into L and R, at least 1/4 of the array values are less than the pivot and at least 1/4 of the array values are larger than the pivot.

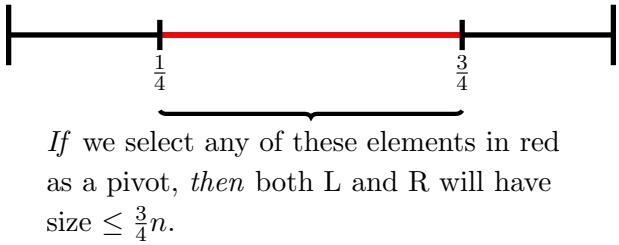


Figure 9: Figure illustrates the region of the data where picking a pivot will result in both L and R having sizes less than $\frac{3}{4}n$.

So, we say that a *phase* in our algorithm ends as soon as we pick a pivot in the middle half of our array, denoted by the red portion in the figure above. Recognize that in phase k, the array size is at most $n\left(\frac{3}{4}\right)^k$. The maximum number of *phases* before we can hit a base case is given by $\lceil \log_{4/3} n \rceil$, since in each phase we dampen the size of the data by $\frac{3}{4}$.

5.4 Analysis - Expected Work

The analysis of this algorithm is as follows. Let us define the number of calls made to `Select` when the size of the input array is of a particular size. That is, let X_k denote the number of times `Select` called with array of input size between

$$n\left(\frac{3}{4}\right)^{k+1} \leq |A| < n\left(\frac{3}{4}\right)^k$$

Total work is given by the sum of the work done for each X_k multiplied by the number of calls of that size. Realize that total work done during phase k given by

$$X_k \cdot cn\left(\frac{3}{4}\right)^k$$

for some constant $c \in \mathbb{R}^+$. Since in each phase, we reduce the input size by at least 3/4, total number of phases given by $\lceil \log_{4/3} n \rceil$. Let W be a random variable describing the total work done.

Then,

$$W \leq \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \left(X_k \cdot cn \left(\frac{3}{4}\right)^k \right) = cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \left(X_k \left(\frac{3}{4}\right)^k \right).$$

We're interested in the expected amount of total work. Hence, we apply the expectation operator to both sides (expectation is a monotone operator), and after applying linearity of expectation we see that, $\mathbb{E}$

$$\mathbb{E}[W] \leq cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \mathbb{E}[X_k] \left(\frac{3}{4}\right)^k.$$

We now analyze $\mathbb{E}[X_k]$. Recall that

$$\mathbb{E}[X_k] = \sum_{i=0}^{\infty} i \cdot \Pr(X_k = i).$$

Claim: X_i is a Geometric Random Variable with parameter $p = \frac{1}{2}$, hence $\mathbb{E}[X_i] = 2$.

Recall that we defined a phase as lasting until we reduce the input size by at least 3/4, Consider our choice of pivot - if the selected pivot is between the 1^{st} and 3^{rd} quartiles of the data, then both L and R will have size $\leq \frac{3}{4}n$. The probability of this occurrence is then $\frac{1}{2}$, so X_i must be a geometric with parameter $\frac{1}{2}$. Put another way, since all pivot choices are independent, $\Pr(X_k = i)$ equals the probability that the first $i-1$ pivot choices were in the outer quartiles *times* the probability that the ith pivot choice was in interquartile range, i.e.

$$\Pr(X_k = i) = \left(\frac{1}{2}\right)^{i-1} \cdot \frac{1}{2} = \frac{1}{2^i}.$$

Hence,

$$\mathbb{E}[X_k] = \sum_{i=0}^{\infty} i \Pr(X_k = i) = \sum_{i=0}^{\infty} \frac{i}{2^i} = 2.$$

That is, in expectation, we need to make two calls to `Select` before reducing our input size by at least 3/4. Ultimately, we see that

$$\mathbb{E}[W] \leq cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \mathbb{E}[X_k] \left(\frac{3}{4}\right)^k \leq 2cn \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k = 2cn \cdot \frac{1}{1 - 3/4} = 8cn = O(n).$$

where we've used the fact that $\sum_k \left(\frac{3}{4}\right)^k$ is a geometric series with constant value (no dependence on n). We have derived expected total work to be $O(n)$. This is in the sequential world. We note that because our recursive calls are stochastic in this algorithm, we cannot use the Master Theorem.

So, by this analysis, in expectation,

$$T_1 = O(n).$$

We note that the work is also $\Theta(n)$ since we have to inspect each element before making a claim on the rank of an element.

Applying Markov's Inequality We may analyze our total work further by utilizing the Markov Inequality. We can compute a bound on the probability that our total work exceeds a multiple of our expected total work. For example, if we wanted to do analysis that our total work will exceed 5 times our expected work:

$$P(\text{Total Work} \geq 5 \times E[\text{Total Work}]) \leq \frac{E[\text{Total Work}]}{5 \times E[\text{Total Work}]} = \frac{1}{5}$$

This lets us say that we will only have do perform $5n$ work with probability at most $1/5$. Call this probability $p = 1/5$. We may run this algorithm as many times as we want. We can run our algorithm $5c2n$ times. Each time, we have at probability at least $4/5$ of finding our kth element in linear time. The total amount of work required is the sum over however many times we have to restart the process (potentially an infinite number of times) *times* the probability we have to restart that many times. Each time we fail, we do $5cn = O(n)$ work. Hence

$$\sum_{r=1}^{\infty} \left(\frac{4}{5}\right)^r O(n) = O(n) \cdot c = O(n).$$

That is, we can try an infinitude of times and yet still only expect to perform constant work.

5.5 Parallelizing our Select Algorithm

Note that there is only one recursive call to **Select**, so the recursive portion of this function is not divide and conquer in nature. How can we make this algorithm run in parallel? It's clear that each step of the algorithm runs in constant time except for creating L and R, hence this is the step we will focus on.

Constructing L and R with Small Depth We seek to construct L and R in a smart way. Naively, you may think that constructing an indicator list of which elements in A go into L and R is enough, but scanning and retrieving a solution from this construction will require $O(n)$ work. We can actually construct these lists using **PrefixSum**, which only takes $O(\log n)$ depth and $O(n)$ work still. We consider the following methodology to keep our depth low when constructing L and R (notice that constructing R follows the same steps as constructing L):

In the first and fourth steps, we request a chunk of $O(n)$ memory in constant time, i.e. we perform $O(n)$ work with unit depth. Also part of this first step, we use n processors and assign each to initialize one element of the array to zero. In our second step, we construct our indicator list B_L which indicates whether an entry belongs in L, i.e. whether the entry less than p. This requires $O(n)$ work but only $O(1)$ depth. Notice that only after completing step 3 do we know how much space to allocate for L (or R), since the value of the last entry in our prefix sum array tells us how many elements belong to L. This third step requires $O(n)$ work and $O(\log n)$ depth. Lastly, we fill in the entries of array L with $O(n)$ work and $O(1)$ depth.

Hence, our algorithm to construct L or R requires $O(n)$ work and $O(\log n)$ depth. So, if we use this (parallel) sub-routine within our `Select` algorithm, the recurrence for depth becomes

Algorithm 9: Constructing L (or R)

```
1 Allocate and empty array of size n, with all values initially 0.
2 Construct indicator list B_L[0, ..., n − 1], where b_i = 1{a_i < p}
3 Compute PrefixSum on B_L                          // This requires O(log n) depth
4 Create the array L of size PrefixSum(B_L)[n − 1]
5 for i = 1, 2, ..., n do
6 |   if B_[i] == 1 then
7 |   |   L[PrefixSum(B_L[i])] ← a[i]                  // Can be done in parallel
8 |   end
9 end
```

$$D(n) = D\left(\frac{n}{4/3}\right) + O(\log n) \implies D(n) = O(\log^2 n).$$

We remark that since our new method for constructing L and R did not require any additional work, our recurrence for work remains the same.

Example Suppose

$$A = \begin{bmatrix} 1 & 16 & 3 & 2 & 5 \end{bmatrix}$$

We randomly select a pivot, suppose we select $p \leftarrow 5$. This takes $O(1)$ work. We assign each element to a single processor, and if the element is strictly smaller than the pivot, we overwrite its corresponding bit in B_L with a 1,

$$B_L = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

This takes $\Theta(n)$ work, but since we perform operations in parallel it requires $O(1)$ depth. Now, we compute the prefix-sum on B_L, which gives us

$$\begin{bmatrix} 1 & 1 & 2 & 3 & 3 \end{bmatrix}$$

i.e. we now know that since B_L has bits turned on in the first, third, and fourth positions, and since the first, third, and fourth elements in the resulting prefix-sum output have values 1, 2, 3, this tells us where to place each element in L. Computing prefix-sum takes $O(n)$ work and $O(\log n)$ depth.

5.6 Parallel Quick Sort

Parallel quick sort is very similar to parallel selection. We randomly pick a pivot and sort the input list into two sublists based on their order relationship with the pivot. However, instead of discarding one sublist, we continue this process recursively until the entire list has been sorted. The pseudocode for this algorithm is as follows:

The analysis for this algorithm is performed in the next lecture.

Algorithm 10: Quicksort(A)

Input : A

Output: sorted A

Let p be random uniformly selected from A

L = Elements of $A < p$

R = Elements of $A > p$

return [**QuickSort**(L), p, **QuickSort**(R)]

6 Memory Management and (Seemingly) Trivial Operations

We have to be extremely careful when analyzing work and depth for parallel algorithms. Even the most trivial operations must be dealt with care. For example, even an operation as trivial as `concatenate`, i.e. in the case where we wish to concatenate two arrays together. Suppose we have two arrays A, B each of length n. If A and B are located in two different places in RAM then we might have a problem. The usual way to deal with these things is to allocate the output array which costs constant time regardless of input size. Then, if we wish to concatenate two arrays into our output, we simply give one thread half the array and the other thread the other half of the array memory. When both threads are done, it's as though we have already performed a concatenate operation for free.

If allowed each thread to return its own array in the usual manner, we can still concatenate the two with $D(n) = O(1)$. To do this, we simply assign each processor one element to be copied or moved in RAM and perform the n operations in parallel. Hence depth is constant. However, we still need to perform $O(n)$ work in this case.

We assume that we can allocate memory in constant time, as long as we don't ask for the memory to have special values in it. That is, we can request a large chunk of memory (filled with garbage bit sequences) in constant time. However, requesting an array of zeros already requires $\Theta(n)$ work since we must ensure the integrity of each entry[24]. In the context of sequential algorithms, this is not a concern since reading in an input of n bits, or outputting n bits already requires $\Theta(n)$ work, so zeroing out an array of size n does not dominate the operation count. However, in some parallel algorithms, no processor reads in the entire input, so naively zeroing out a large array can easily dominate the operation time of the algorithm.

7 QuickSort

With this in mind, we will now finish the analysis of QuickSort, taking a closer look at memory management dueing the algorithm. The algorithm is as follows.

Algorithm 11: QuickSort

Input: An array A

Output: Sorted A

1 $p \leftarrow$ element of A chosen uniformly at random

2 $L \leftarrow [a | a \in A \text{ s.t. } a < p]$ `// Implicitly:` $B_L \leftarrow \mathbb{1}\{a_i < p\}_{i=1}^n$, `prefixSum`$(B_l)$,

3 $R \leftarrow [a | a \in A \text{ s.t. } a > p]$ `// which requires` $\Theta(n)$ `work and` $O(\log n)$ `depth.`

4 **return** [QuickSort(L), p, QuickSort(R)]

[24] In practice, zeroing out an array is optimized by hardware and is not a concern.

7.1 Analysis on Memory Management

Recall that in Lecture 4, we designed an algorithm to construct L and R in $O(n)$ work and $O(\log n)$ depth. We will take this opportunity to highlight some of the intricacies of memory management during the construction of L and R.

Selecting a pivot uniformly at random We denote the size our input array A by n. To be precise, we can perform step 1 in $\Theta(\log n)$ work and $O(1)$ depth. That is, to generate a number uniformly from the set $\{1, 2, \ldots, n\}$ we can assign $\log n$ processors to independently flip a bit "on" with probability $1/2$. The resulting bit-sequence can be interpreted as a $\log_2$ representation of a number from $\{1, \ldots, n\}$.

Allocating storage for L and R Start by making a call to the OS to allocate an array of n elements; this requires $O(1)$ work and depth, since we do not require the elements to be initialized. We compare each element in the array with the pivot, p, and write a 1 to the corresponding element if the element belongs in L (i.e. it's smaller) and a 0 otherwise. This requires $\Theta(n)$ work but can be done in parallel, i.e. $O(1)$ depth. We are left with an array of 1's and 0's indicating whether an element belongs in L or not, call it $\mathbb{1}_L$,

$$\mathbb{1}_L = \mathbb{1}\{a \in A \text{ s.t. } a < p\}.$$

We then apply `PrefixSum` on the indicator array $\mathbb{1}_L$, which requires $O(n)$ work and $O(\log n)$ depth. Then, we may examine the value of the last element in the output array from `prefixSum` to learn the size of L. Looking up the last element in array $\mathbb{1}_L$ requires $O(1)$ work and depth. We can further allocate a new array for L in constant time and depth. Since we know $|L|$ and we know n, we also know $|R| = n - |L|$; computing $|R|$ and allocating corresponding storage requires $O(1)$ work and depth.

Thus, allocating space for L and R requires $O(n)$ work and $O(\log n)$ depth.

Filling L and R Now, we use n processors, assigning each to exactly one element in our input array A, and in parallel we perform the following steps. Each processor $1, 2, \ldots, n$ is assigned to its corresponding entry in A. Suppose we fix attention to the kth processor, which is responsible for assigning the kth entry in A to its appropriate location in either L or R. We first examine $\mathbb{1}_L$`[k]` to determine whether the element belongs in L or R. In addition, examine the corresponding entry in `prefixSum` output, denote this value by $i =$ `prefixSum`$(\mathbb{1}_L)$`[k]`. If the kth entry of A belongs in L, then it may be written to the position i in L immediately, by definition of how what our `prefixSum` output on $\mathbb{1}_L$ means. If the kth entry instead belongs in R, then realize that index i tells us that exactly i entries "before" element k belong in L. Hence exactly $k - i$ elements belong in array R before element $A[k]$. Hence we know exactly where to write the kth element to R if it belongs there.

Clearly, the process of filling L and R requires $O(n)$ work and $O(1)$ depth.

Work and Depth per Iteration Thus we may say that steps 1,2,3 of our algorithm require $O(n)$ work and $O(\log n)$ depth.[25] The last step of the algorithm requires recursive calls to `Quicksort` and a concatenation of several elements. Realize that if we are clever about how we allocate the memory for our arrays to begin with, this "concatenation" (or lack thereof) can be performed without extra work or depth.[26]

7.2 Total Expected Work

Now we analyze the total expected work of QuickSort.[27] We assume all our input elements are unique.[28] On a very high level, to compute the work, note that the work done summed across each level of our computational DAG is n. There are $\log_{4/3} n$ levels in our DAG, hence expected work given by

$$\mathbb{E}[T_1] = \mathbb{E}[\#\text{ levels}] \cdot \mathbb{E}[\text{work per level}] = O(n \log n)$$

Details We define the random indicator variable X_{ij} to be one if the algorithm *does compare* the ith *smallest* and the jth *smallest* elements of input array A during the course of its sorting routine, and zero otherwise. Let X denote the *total* number of comparisons made by our algorithm. Then, we have that

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Number of Comparisons We take a moment to realize why we sum over $\binom{n}{2}$ elements instead of n^2: the only time in our `QuickSort` algorithm whereby we make a comparison of two elements is when we construct L and R. In doing so, we compare each element in the array to a fixed pivot of p, after which all elements in L less than p and all elements in R greater than p. Realize that pivot p is never compared to elements in L and R for the remainder of the algorithm. Hence each of the $\binom{n}{2}$ pairings are considered *at most* once by our `QuickSort` algorithm.

Expected Number of Comparisons Realize that expectation operator $\mathbb{E}$ is monotone, hence

$$\mathbb{E}[X] \leq \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}],$$

[25] It may be tempting to think that we can get away with *not* calculating a `prefixSum` on $\mathbb{1}_L$ in order to determine the sizes of L and R, and instead pass this indicator array $\mathbb{1}_L$ to our recursive call. We might think that we can then avoid incurring $\log n$ depth. However, realize that then when we pass $\mathbb{1}_L$ to our recursive call, it would still be of size n, hence we would not be decreasing the size of our problem with each recursive call. This would result in quite a poor algorithm.

[26] For details, see lecture 4 notes, specifically the section on "Parallel Algorithms and (Seemingly) Trivial Operations".

[27] We follow the analysis from CMU very closely.[2]

[28] If there are duplicate elements in our input A, then this only cuts down the amount of work required. As the algorithm is written, we look for strict inequality when constructing L and R. If we ever select one of the duplicated elements as a pivot, its duplicates values are not included in recursive calls hence the size of our sub-problems decreases even more than if all elements were unique.

where the equality follows from linearity of expectation. Since X_{ij} an indicator random variable,

$$\mathbb{E}[X_{ij}] = 0 \cdot \Pr(X_{ij} = 0) + 1 \cdot \Pr(X_{ij} = 1) = \Pr(X_{ij}).$$

Consider any particular X_{ij} for $i < j$ (i.e. one of interest). Denote the ith smallest element of input array A by $\text{rank}_A^{-1}(i)$ for $i = 1, 2, \ldots, n$. For any one call to `QuickSort`, there are exactly *three* cases to consider, depending on how the value of the pivot compares with $A[i]$ and $A[j]$.

- **Case 1: $p \in \{\text{rank}_A^{-1}(i), \text{rank}_A^{-1}(j)\}$** In selecting a number from $\{1, 2, \ldots, n\}$ uniformly at random, we happen to select an element as pivot from A which is either the ith smallest element in A or the jth smallest element in A. In this case, $X_{ij} = 1$ by definition.
- **Case 2: $\text{rank}_A^{-1}(i) < p < \text{rank}_A^{-1}(j)$**: The value of the pivot element selected lies between the ith smallest element and the jth smallest element. Since $i < j$, element `i` placed in L and `j` placed in R and $X_{ij} = 0$, since the elements will never be compared.
- **Case 3: $p < \text{rank}_A^{-1}(i)$ or $p > \text{rank}_A^{-1}(j)$**: The value of the pivot is either less than the ith smallest element in A or greater than the jth smallest, in which case either both elements placed into R or both placed into L respectively. Hence X_{ij} may still be "flipped on" in another recursive call to `Quicksort`.

It's possible that on any given round of our algorithm, we end up falling into case 3. Ignore this for now. In doing so, we implicitly condition on falling into case 1 or 2, i.e. we condition on our rank p being chosen so that it lies in the set of values $\{\text{rank}_A^{-1}(i), \text{rank}_A^{-1}(i+1), \ldots, \text{rank}_A^{-1}(j)\}$. Then, the probability that $X_{ij} = 1$ (ignoring case 3) is exactly

$$2/(j - i + 1),$$

Thus,

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} 2 \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\ &= \sum_{i=1}^{n-1} 2 \left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1} \right).\end{aligned}$$

Now, note that $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the Harmonic Number. We note that $\sum_{i=1}^{n} \frac{1}{i} < \int_1^n \frac{1}{x} \mathrm{d}x$, from Calculus,[29] and further that $\int_1^n \frac{1}{x} \mathrm{d}x = \ln n$. Hence we see that

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} 2 \sum_{j=2}^{n-i+1} \frac{1}{j} < 2nH_n = O(n \log n).$$

So, in expectation, the number of comparisons (i.e. the work performed by our algorithm) is $O(n \log n)$.

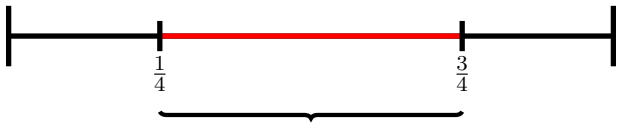
[29] This can easily be seen by comparing a Lower Darboux Sum with a Riemann Integral.

7.3 Total Expected Depth

Recall our analysis in Lecture 4 for `QuickSelect`. In a similar fashion, we may consider the number of *recursive calls* made to our algorithm when our input array is of size

$$\left(\frac{3}{4}\right)^{k+1} n < |A| < \left(\frac{3}{4}\right)^{k} n,$$

and denote this quantity by X_k. We saw that if we select a pivot in the right way, we can reduce input size by 3/4.



If we select any of these elements in red as a pivot, where we visualize the elements being in sorted order, *then* both L and R will have size $\leq \frac{3}{4}n$.

Figure 10: Ideal Pivot Selection

We saw that $\mathbb{E}[X_k] = 2$, since it's a geometric random variable (i.e., we continually make calls to `QuickSort` until we successfully reduce our input size by at least 3/4. In expectation, this takes two trials. Hence in expectation we require $2c \log_{4/3} n$ recursive calls to be made before we reach a base case.

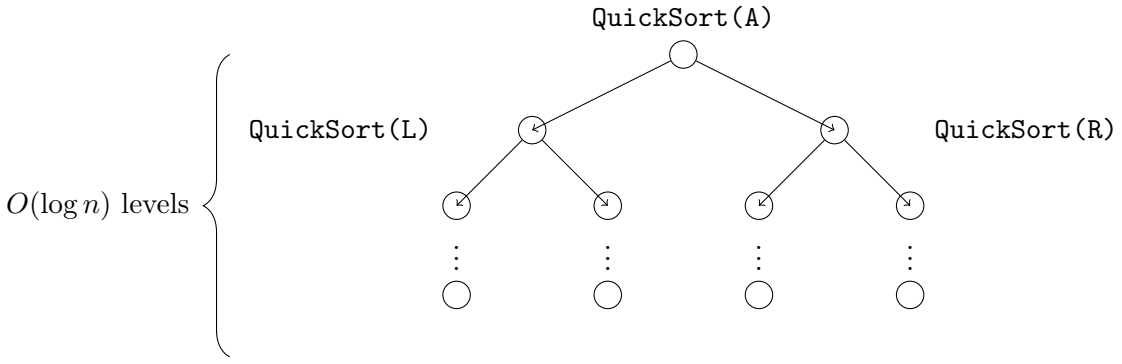


Figure 11: Computational DAG for `QuickSort`

Here, we see that in our computational DAG, we have $O(\log_{4/3} n)$ levels. At each level, our algorithm requires $O(\log n)$ depth since the bottleneck is in constructing L and R and specifically in the call to Prefix Sum. Hence in expectation our total depth given by the expected number of levels times the expected depth per level: $\mathbb{E}[D(n)] = O(\log^2 n)$.

7.4 A Shortcut for Bounding Total Expected Work

Examine figure 7.3, the computational DAG for our QuickSort algorithm. We are initially handed an input array of size n elements. At each level of our DAG, the n elements are split into chunks across nodes in that level. Specifically, we're not sure exactly how many elements from A are allocated to L, nor do we know exactly how many elements from A are allocated to R. But we definitely know that there are always n elements we deal with on each level. The most work-intensive operation we perform in each call to QuickSort is in constructing L and R, which requires $\Theta(n)$ work. Hence, for a given level in our computational DAG, we perform exactly $\Theta(n)$ work total after summing across work in all nodes.

So, in this *very specialized analysis*, we have that expected work is

$$\mathbb{E}[W] = (\#\text{ nodes per level}) \times (\#\text{ levels}) = O(n \log n).$$

Since there are $O(\log n)$ levels in our tree in expectation, we perform $O(n \log n)$ work in expectation.

8 Matrix multiplication: Strassen's algorithm

We've all learned the naive way to perform matrix multiplies in $O(n^3)$ time.[30] In today's lecture, we review Strassen's sequential algorithm for matrix multiplication which requires $O(n^{\log_2 7}) = O(n^{2.81})$ operations; the algorithm is amenable to parallelization.[9]

A variant of Strassen's sequential algorithm was developed by Coppersmith and Winograd, they achieved a run time of $O(n^{2.375})$.[6] The current best algorithm for matrix multiplication $O(n^{2.373})$ was developed by Stanford's own Virginia Williams[10].

8.1 Idea - Block Matrix Multiplication

The idea behind Strassen's algorithm is in the formulation of matrix multiplication as a recursive problem. We first cover a variant of the naive algorithm, formulated in terms of block matrices, and then parallelize it. Assume $A, B \in \mathbb{R}^{n \times n}$ and $C = AB$, where n is a power of two.[31]

We write A and B as block matrices,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

where block matrices A_{ij} are of size $n/2 \times n/2$ (same with respect to block entries of B and C). Trivially, we may apply the definition of block-matrix multiplication to write down a formula for the block-entries of C, i.e.

[30] Refresher, to compute $C = AB$, we need to compute c_{ij}, of which there are n^2 entries. Each one may be computed via $c_{ij} = \langle a_i^T, b_j \rangle$ in $2n - 1 = \Theta(n)$ operations. Hence total work is $O(n^3)$.

[31] If n is *not* a power of two, then from a theoretical perspective we may simply pad the matrix with additional zeros. From a practical perspective, we would simply use un-equal size blocks.

$$
\begin{aligned}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{aligned}
$$

8.2 Parallelizing the Algorithm

Realize that A_{ij} and $B_{k\ell}$ are smaller matrices, hence we have broken down our initial problem of multiplying two $n \times n$ matrices into a problem requiring 8 matrix multiplies between matrices of size $n/2 \times n/2$, as well as a total of 4 matrix additions. There is nothing fundamentally different between the matrix multiplies that we need to compute at this level relative to our original problem.

Further, realize that the four block entries of C may be computed independently from one another, hence we may come up with the following recurrence for *work:*

$$W(n) = 8W(n/2) + O(n^2)$$

By the Master Theorem,[32] $W(n) = O(n^{\log_2 8}) = O(n^3)$. So we have *not* made any progress (other than making our algorithm parallel). We already saw in lecture two that we can naively parallelize matrix-multiplies very simply to yield $O(n^3)$ work and $O(\log n)$ depth.

8.3 Strassen's Algorithm

We now turn toward Strassen's algorithm, such that we will be able to reduce the number of sub-calls to matrix-multiplies to 7, using just a bit of algebra. In this way, we bring the work down to $O(n^{\log_2 7})$.

How do we do this? We use the following factoring scheme. We write down C_{ij}'s in terms of block matrices M_k's. Each M_k may be calculated simply from products and sums of sub-blocks of A and B. That is, we let

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})\,(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

[32] Case 1: $f(n) = O(n^2)$, so $c = 2 < 3 = \log_2(8)$.

Crucially, each of the above factors can be evaluated using exactly *one* matrix multiplication. And yet, since each of the M_k's expands by the distributive property of matrix multiplication, they capture additional information. Also important, is that these matrices M_k may be computed independently of one another, i.e. this is where the parallelization of our algorithm occurs.

It can be verified that

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M3 + M_6 \end{aligned}$$

Realize that our algorithm requires quite a few summations, however, this number is a constant independent of the size of our matrix multiples. Hence, the work is given by a recurrence of the form

$$W(n) = 7W(n/2) + O(n^2) \implies W(n) = O(n^{\log_2 7}).$$

What about the depth of this algorithm? Since all of our recursive matrix-multiplies may be computed in parallel, and since we can add matrices together in unit depth,[33] we see that depth is given by

$$D(n) = D(n/2) + O(1) \implies D(n) = O(\log n)$$

By Brent's theorem, $T_p \leq \frac{n^{2.81}}{p} + O(\log n)$. In the years since Strassen published his paper, people have been playing this game to bring down the work required marginally, but nobody has come up with a fundamentally different approach.

8.4 Drawbacks of Divide and Conquer

We now discuss some bottleneck's of Strassen's algorithm (and Divide and Conquer algorithms in general).

- We haven't considered communication bottlenecks; in real life communication is expensive.
- Disk/RAM differences are a bottleneck for recursive algorithms, and
- PRAM assumes perfect scheduling.

[33] We note that to perform matrix addition of two $n \times n$ matrices $X + Y = W$, we may calculate each of the n^2 entries $W_{ij} = X_{ij} + Y_{ij}$ in parallel using n^2 processors. Each entry requires only one fundamental unit of computation, hence the work for matrix addition is $O(n^2)$ and the depth is $O(1)$.

Caveat - Big $\mathcal{O}$ and Big Constants One last caveat specific to Strassen's Algorithm is that in practice, the $O(n^2)$ term requires $20 \cdot n^2$ operations, which is quite a large constant to hide. If our data is large enough that it must be distributed across machines in order to store it all, then really we can often only afford to pass through the entire data set one time. If each matrix-multiply requires twenty passes through the data, we're in big trouble. Big $\mathcal{O}$ notation is great to get you started, and tells us to throw away egregiously inefficient algorithms. But once we get down to comparing two reasonable algorithms, we often have to look at the algorithms more closely.

When is Strassen's worth it? If we're actually in the PRAM model, i.e. we have a shared memory cluster, then Strassen's algorithm tends to be advantageous only if $n \geq 1,000$, *assuming no communication costs*. Higher communication costs drive up the n at which Strassen's becomes useful very quickly. Even at $n = 1,000$, naive matrix-multiply requires $1e9$ operations; we can't really do much more than this with a single processor. Strassen's is mainly interesting as a theoretical idea. For more on Strassen in distributed models, see [1].

Disk Vs. RAM Trade-off What is the reason that we can only pass through our data once? There is a big trade-off between having data in ram and having it on disk. If we have tons of data, our data is stored on disk. We also have an additional constraint that with respect to *streaming data*, as the data are coming in they are being stored in memory, i.e. we have fast random access, but once we store the data to disk retrieving it again is expensive.

Lecture contents

1. Graph contractions
2. Star contraction
3. Parallel connected components (random mates)
4. Intro to optimization

9 Minimum spanning tree algorithms

Now we'll shift our focus to parallel graph algorithms, beginning with minimum spanning trees. In the following sections, we'll denote our *connected* and *undirected* graph by $G = (V, E, w)$. The size of the vertex set $|V| = n$, the size of the edge set $|E| = m$, and we assume that the weights $w : E \to \mathbb{R}$ are distinct for convenience.[34]

A *tree* is an undirected graph G which satisfies any two of the three following statements, such a graph also then satisfies the other property as well.[35]

1. G connected
2. G has no cycles
3. G has exactly $n - 1$ edges

The *minimum spanning tree* (MST) of G is $T^* = (V, E_T, w)$ such that $\sum_{e \in E_{T^*}} w(e) \leq \sum_{e \in E_T} w(e)$ for any other spanning tree T. Under the assumptions on G, the MST T^* is unique and the previous inequality is strict.

9.1 Sequential approaches and Kruskal's algorithm

Sequential algorithms for finding the MST are nice and easy because greedy algorithms work well for this problem. The two clear approaches are Prim's algorithm and Kruskal's algorithm. Here, we'll focus on Kruskal's algorithm, given in Algorithm 12.

Kruskal's algorithm is based on the *cut property* (or *red rule*), which we now prove.

Theorem 9.1 (Cut-Property, Red-Rule) *Let $G(V, E, w)$ be an undirected and connected graph. Then the edge with minimum weight leaving any cut $S \subset V$ is in the MST of G.*

[34] If the edge weights are not distinct, then we may simply perturb each edge weight by a small random factor $\epsilon > 0$, where $\epsilon < 1/n^3$. We no longer have to break ties in our algorithm and at the end, we may simply round back the edge weights to recover the weight of the minimum spanning tree.

[35] To see that $(1, 2) \implies 3$, use induction on the number of nodes n. The base case is trivial. For the inductive step, realize that any acyclic connected graph G must have a leaf v where $d(v) = 1$. Since $G - v$ also acyclic and connected, by the induction hypothesis, $e(G') = n - 2$. Since $d(v) = 1$, we have that $e(G) = n - 1$. To see that $(1, 3) \implies 2$, suppose toward contradiction G has a cycle. To see that $(2, 3) \implies 1$, consider the case that G split into k components. Since G has no cycles, each component G_i both connected and acyclic, hence each is a tree. So total number of edges in G given by $\sum_i (n(G_i) - 1) = n - k$. But since $|E| = n - 1$, $k = 1$.

Algorithm 12: Kruskal's MST algorithm

Input: $G = (V, E, w)$, an undirected and connected graph
Result: T^*, the MST of G

Sort E in increasing order by edge weight $T \leftarrow \emptyset$
while T *not yet a spanning tree* **do**
  $e \leftarrow$ next edge in the queue `// i.e. lightest edge yet to be considered`
  if $(T \cup \{e\})$ *contains no cycles* **then**
    $T \leftarrow T \cup \{e\}$ `// By red rule,` e `belongs in` T^*.
  end
end
return T

Proof: We are given a graph G and a cut $S \subset V$. We say that an edge is in a cut S if exactly one incident node is in S and the other incident node of the edge in $V \setminus S$. Let $e^* = (u, v)$ be the minimum weight edge in cut S. Assume toward contradiction that the edge with minimum weight leaving cut S is *not* in the MST T, i.e. that $e^* \notin T$ where T is the MST of G and

$$e^* = \arg\min_{e \in S} w(e).$$

Suppose $e^* = (u, v)$. Since T is a spanning tree of G, we know that u and v are connected in the edge set of T, i.e. there exists a u-v path in T. Of course, we have assumed that $(u, v) \notin T$. We construct a u-v path using depth first search. Let $(x, y) = e'$ denote the edge in T which crosses cut S in the u-v path. Replace (x, y) with (u, v). Call the result T'.

We claim that $T' = (T \setminus (x, y)) \cup (u, v)$ is still a tree. We first claim T' retains connectivity among all nodes. To see this, realize first that since (x, y) and (u, v) each have exactly one incident node in S and one incident node in $V \setminus S$, then it is *not* possible that either (x, y) or (u, v), when removed from T, could result in S becoming disconnected or $V \setminus S$ becoming disconnected. Hence, without either of these edges, we can get from all nodes in S to all other nodes in S, and same goes for the set $V \setminus S$. It remains to show that we can still traverse from S to $V \setminus S$. Realize that all paths previously going from S to $V \setminus S$ using (x, y) can now simply utilize (u, v) instead.

Further, realize that we have not created any cycles. Specifically, when we add edge (u, v) to the tree T, we induce a cycle. But realize that the edge (x, y) is part of this cycle, and we immediately remove it. Hence in maintaining connectivity and keeping exactly $n - 1$ edges, by definition of a tree we know that the result is acyclic.

Since e^* is the minimum weight edge in S, and since it is not contained in T, we know that $w(e^*) < w(e')$. But then this implies that

$$w(T') = \sum_{e \in T'} w(e) = \sum_{e \in T} w(e) - \underbrace{[w(e') - w(e^*)]}_{>0} < \sum_{e \in T} w(e) = w(T)$$

But this is a contradiction, since T was defined as the MST of G. Thus, e^* is in the MST of G. ■

9.2 Complexity Analysis for Kruskal's

A brief complexity analysis of Kruskal's algorithm is as follows.[36] Recall work is defined as $T_1 = W(n)$.

Sorting Edges Our QuickSort algorithm requires $\mathbb{E}[W(n)] = O(m \log m) = O(m \log n)$.

Checking for Cycles Specifically because we are considering adding a single edge to a tree, we can check if we are inducing a cycle in almost constant time; specifically the work required is $\alpha(m, n)$, where α denotes the Inverse Ackermann function.[4]

We show now quickly a way to check for cycles in $O(\log n)$ work that is a bit more accessible. We must ensure that if edge $e = (u, v)$ to be added, that the component of u is *not* the same component as v. To do this, we keep several data structures around: an array and a Binary Search Tree. We assume that with each component of our graph, we associate with it a binary search tree storing values of nodes in our graph G which are in a particular component k. In addition, we keep an array of length n where in each entry, it tells us which component node i currently in.

Hence, given a candidate (u, v), we go to our length n array and look up in constant time $a[v]$ which tells us that node v in component j. We then go to our BST corresponding to component j, and search in $O(\log n)$ time to check if node u in the same component.

Updating Data Structures if we Add an Edge If we find they share the same component already, we discard the edge and continue. Otherwise, the edge turns out to connect two components, hence we must update our data structures. We go to array A of length n, and write down that u now belongs to v's component.[37] This last step requires $O(1)$ work.

Now, we must merge the component of u with the component of v. Two merge two binary search trees, we first flatten each BST into a sorted linked list with $O(n)$ work using in order traversal. We then merge two sorted lists in $O(n)$ work to get a sorted *array*. We then need to form a BST with our sorted array. To do this, we fix attention to the element in the middle of our array. Realize that all preceding elements in the list are no larger than itself, and all elements following are no smaller. We may make the same observation for the left and right partitions, each time storing pointers from the median element of the list to the left and right children medians. In this way, we re-construct our BST with $O(n)$ work, since we only end up applying a constant amount of work to each element in our sorted array.

Total Work Sorting edges costs $O(m \log n)$ work. Our while loop iterates over at most m edges. Checking for cycles each iteration costs $O(\log n)$ work. Hence we have incurred $O(m \log n)$ work

[36] It's worthwhile to note that since $\binom{n}{2} \leq m \implies m = O(n^2)$, hence any $O(\log m)$ term may actually be replaced by $O(\log n)$, using the fact that $\log m = O(\log n^2) = O(2 \log n) = O(\log n)$. We leave $O(\log m)$ terms in place here for pedagogical purposes.

[37] We resolve the conflict of which node to join to which component by (arbitrarily) choose node index v as the parent since it is larger than node index u.

thus far without even considering how much it costs to update our data structures when we add an edge to our tree.

Realize that we only need to add $n-1$ edges before we construct a tree. Further, each edge we add requires $O(n)$ work. Hence the entire tree construction process takes $O(n^2)$ work. Since $O(n^2) = O(m)$, the work required to sort our edges dominates, and total work required is $O(m \log n)$.

10 Parallel MST via Boruvka's Algorithm

Since we need to evaluate the edges in order of weight, Kruskal's algorithm is inherently sequential. We need a new approach if we want a parallel MST algorithm. Kruskal's algorithm is instructive in that it uses the Cut Property which we proved last lecture. Boruvka's algorithm, will also use this property. [7]

10.1 Observation - Smallest Weight Incident Edge on each Node belongs in MST

A single node is a subset of the nodes, i.e. each node can be considered individually to define a cut. And so if you look at a single node and its incident edges, the smallest one must be in the MST by applying the cut property. Realize that if we determined the smallest weight edge incident to *each* node in our graph, we *must* find at least $n/2$ unique edges belonging to the MST. Why $n/2$? When we search for a lowest weight incident edge on each node, realize that it's possible the node to which such an edge connects also selects the same edge for inclusion in the MST. At worst, this could happen for each node. Hence we must find at least $n/2$ edges in our MST.

10.2 Edge Contraction

Before we can recursively apply our algorithm, we must first take each of the (at least) $n/2$ edges we found belonging to the MST and "merge" the two incident nodes into one.

Deleting Duplicate Edges Suppose we add edge (u, v) to our tree. The neighborhood of u and the neighborhood of v may overlap, hence we have two ways to get from super-node u-v to such a neighbor t. When we contract edge (u, v), we would yield a multi-graph, in which two vertices may be connected by more than one edge. We wish to avoid this, and since our algorithm uses the Red-Rule (and is interested in the lowest weight edge in a cut), we can ignore the edge with larger weight. That is, if $\Gamma(u) \cap \Gamma(v) \neq \emptyset$ in our original graph G, then for each node t in the intersection of these two neighborhoods, we compare $w(u, t)$ with $w(v, t)$ and delete the edge with larger weight.

Recursive formulation But notice that when we contract the edges, the resulting "super" node (consisting of multiple nodes) itself defines a cut. Hence we can again apply the Red-Rule and find the smallest weight edge incident to each of the nodes in the *contracted* graph, and again continue to find more edges belonging to the MST. Hence we repeat the algorithm until we realize an MST.

Since each time we find at least $n/2$ edges, we only must repeat our algorithm at most $\log_2 n$ times before finding an MST.

10.3 Boruvka's Algorithm

We assume that the graph is stored in an Adjacency-List, i.e. for each node we store its neighbors in a linked list. Since each node can have at most $n-1$ neighbors, each adjacency list can have at most $n-1$ entries. There are n adjacency lists (one for each node).

Algorithm 13: Boruvka's Algorithm

Input : Graph G, represented by adjacency list

1 **for** *each node, in parallel* **do**

2 | Compute smallest weight incident edge `depth` // $O(m)$ `work,` $O(\log n)$

3 **end**

4 Contract edges

5 Merge Adjacency Lists // $O(n)$ `work and` $O(1)$ `depth`

6 Recurse

10.4 Analysis

10.4.1 Finding Smallest Weight Incident Edges

Examine the initial `for` loop which is executed in parallel. We know that we're looking for smallest weight edges, hence we know that we need to examine each edge at least once, i.e. we require at least $O(m)$ work. By the handshake lemma, $\sum_i \deg(v_i) = 2|E|$. Since for each node $v \in V$, we need to find the minimal weight incident edge, we must scan through $2|E|$ neighbors in total. We may use our parallel quickSelect algorithm on each adjacency list which requires $O(\deg(v_i))$ work and $O(\log \deg(v_i))$ depth per node. To bound total work, we must sum across all nodes, hence total work is $O(m)$. For depth, we are concerned with $\max_i \deg(v_i)$, which is $O(n)$. Hence depth is $O(\log n)$.

10.4.2 Contracting Edges

Now that we've shown that we remove $n/2$ isolated vertices in each round, we need to analyze how many edges we remove in each iteration. This will give a full picture of how fast are our subproblems decreasing in size. Notice that the number of edges removed in a contraction is *at least* equal to the number of vertices (since each vertex selects a lowest weight incident edge). Consider a best case possible sequence of events.

round	vertices	edges
1	n	m
2	$n/2$	$m - n/2$
3	$n/4$	$m - n/2 - n/4 = m - 3n/4$
$\vdots$	$\vdots$	$\vdots$
k	$n/2^{k-1}$	$m - (2^{k-1} - 1)n/(2^{k-1})$

Notice that for all $k \in \mathbb{N}$,

$$n\frac{2^{k-1}-1}{2^{k-1}} = n\underbrace{\left(1 - \frac{1}{2^{k-1}}\right)}_{<1}$$

hence the number of edges never quite drops below $m - n$. So as long as there are $m > 2n$ edges to start with, work is $O(m \log n)$.

Crucial Observation: Minimum Weight Edges form a Forest Crucially, in the first stage of our algorithm, each node its own connected component. It's also a tree, since a component of size 1 with 0 edges is connected and acyclic hence it's a tree. Realize that when we expand via minimum weight edges, since these edges are guaranteed to be in the MST, they *cannot* form a cycle, hence the result we get is a *forest*. Hence, our job is now to actually contract an entire tree (such that we may apply this to each tree in the forest).

To contract a tree, have each node (in parallel) flip a coin. If a vertex flips heads, it becomes the *center* of a star. If it flips tails, then the vertex attempts to become a *satellite* of (some) star by finding a neighbor which is a center.[38] If no such neighbor exists, i.e. if all other neighbors flipped tails or the vertex is isolated, then the vertex becomes a center.

When we perform a contraction, we need to select one node as the center, and the rest are satellites. Edges between satellites and centers must be deleted. Edges which cross partitions must be kept.[39]

Claim 10.1 *For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of satellites in a call to our tree contraction process described above. Then,*

$$\mathbb{E}[X_n] \geq n/4.$$

Proof: Consider any non-isolated vertex $v \in V(G)$. By definition, a non-isolated vertex has at least one neighbor, hence the probability that v becomes a satellite is *at least* Pr(node v flips Tails) $\times$ Pr(neighbor flips Heads) $\geq \frac{1}{4}$. Hence by linearity of expectation,

$$\mathbb{E}[\#\text{ Isolated Vertices}] = \sum_{\text{non-isolated vertices}} \mathbb{E}[\text{vertex } i \text{ becomes isolated}] \geq n/4.$$

■

[38] If multiple a vertex has multiple neighbors which are centers, it may select one arbitrarily. For example, we select the one with lowest node index.

[39] We now turn to Lemma 16.17 of Blelloch and Maggs.

Contracting a tree yields another Tree Since when doing start contraction on a tree, it remains a tree on each step, the number of edges does down with the number of vertices. Further, since a tree on n nodes has $n-1$ edges, the number of edges in a forest is never more than n. Hence the number of edges in our graph decrease geometrically in expectation. Hence total expected work given by

$$\mathbb{E}[W(n,m)] = \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i kn = O(n).$$

The *Depth* of the contraction is given by $\mathbb{E}[D(n,m)] = O(\log n)$ since when we accumulate *neighborhood lists*, adding an edge to it requires $O(\log n)$ depth since the height of the neighbor hood tree is at most $O(\log n)$.

10.4.3 Merge Adjacency Lists

In terms of our data structures, we have n adjacency lists, and if we are contracting edge (u, v), then we must merge their corresponding adjacency lists together. Suppose that our `list` data structure is singly linked with a head to both the head and tail. Since $u < v$ via lexicographical comparison, we choose to make u the "parent" node, i.e. v's neighbors are absorbed into the neighborhood of u. Hence, we may simply take the tail of u's adjacency list and point it to the head of v's. This takes $O(1)$ work.

Since we must contract at least $n/2$ edges each iteration, this requires $O(n)$ work total. Notice that we may contract each edge independently of the rest, hence we have $O(1)$ depth.

10.4.4 Total Work and Depth

Total Work Notice that our algorithm employs *unary tail recursion*. We have a chain of $O(\log n)$ recursive calls to our algorithm. At each recursive call, we require $O(m)$ work to be performed, since our bottleneck is in finding the smallest weight edge incident each component. Hence total work simply $W(n,m) = O(m \log n)$.

Total Depth Note that at each of the $O(\log n)$ recursive calls, we require $O(\log n)$ depth in order to perform a min-scan. Hence total depth given by $D(n,m) = O(\log^2 n)$.

11 Iterative Solutions for Solving Systems of Linear Equations

First we will introduce a number of methods for solving linear equations. These methods are extremely popular, especially when the problem is large such as those that arise from determining numerical solutions to linear partial differential equations.

The objective for solving a system of linear of equations is as follows. Let A be a full-rank matrix in $\mathbb{R}^{n\times n}$, and b be a vector in $\mathbb{R}^n$. The objective is to find x that satisfies

$$Ax = b.$$

We are guaranteed that x is unique because we assumed A to be invertible. The key observation is that the components of x can be solved for in terms of each other. Specifically, if $a_{ii} \neq 0$ and all of x_j, $j \neq i$ are known then we can solve for x_i as

$$x_i = -\frac{1}{a_{ii}} \left(\sum_{j\neq i} a_{ij} x_j - b_i \right).$$

If this is true for all i then we can solve for each x_i in parallel. Clearly, this cannot be the case (or we would have the solution!), however if have *estimates* for each component, we can solve for new estimates of the components in simultaneously. The algorithms introduced in this section work by iteratively computing estimates of the solution. If each estimate is closer to the true solution than the previous one, then we will converge to the true solution. Each algorithm is initialized with any $x(0) \in \mathbb{R}^n$, and the next iterates are computed as follows.

The Jacobi Algorithm applies this idea directly, by simply using the estimate of x at iteration t, x_t, to compute the estimate of x at time $t+1$, x_{t+1}. The iterates in the Jacobi algorithm are given as follows.

Jacobi Algorithm

$$x_i(t+1) = -\frac{1}{a_{ii}} \left(\sum_{j\neq i} a_{ij} x_j(t) - b_i \right).$$

The Gauss-Seidel algorithm boosts convergence by using information as soon as it is computed. Specifically, if $x_i(t+1)$ is computed before a processor begins computing $x_j(t+1)$, then the Gauss-Seidel algorithm uses $x_i(t+1)$ in place of $x_i(t)$. The iterates in the Gauss-Seidel algorithm are given as follows.

Gauss-Seidel Algorithm

$$x_i(t+1) = -\frac{1}{a_{ii}} \left(\sum_{j<i} a_{ij} x_j(t+1) + \sum_{j>i} a_{ij} x_j(t) - b_i \right).$$

The choice to update the coordinates in order is arbitrary, and in general, different update orders may produce substantially different results. We will see later how the order of updates can significantly impact how parallelizable the Gauss-Seidel algorithm is.

There are also relaxed versions of these algorithms that weight the previous iterate and the Jacobi and Gauss-Seidel updates respectively.

Jacobi Overrelaxation

$$x_i(t+1) = (1-\gamma) x_i(t) - \frac{\gamma}{a_{ii}} \left(\sum_{j \neq i} a_{ij} x_j(t) - b_i \right).$$

Successive Overrelaxation

$$x_i(t+1) = (1-\gamma) x_i(t) - \frac{\gamma}{a_{ii}} \left(\sum_{j<i} a_{ij} x_j(t+1) + \sum_{j>i} a_{ij} x_j(t) - b_i \right).$$

Richardson's Method Richardson's method is obtained by rewriting $Ax = b$ as $x = x - \gamma[Ax - b]$. The updates are then given by

$$x(t+1) = [I - \gamma A] x(t) + \gamma b$$

where γ is a scalar relaxation parameter. Richardson's method can also be executed in a Gauss-Seidel fashion (called RGS). Explicitly, the updates for the RGS algorithm are given by

$$x_i(t+1) = x_i(t) - \gamma \left[\sum_{i<j} a_{ij} x_j(t+1) + \sum_{j \geq i} a_{ij} x_j(t) - b_i \right]$$

Here we emphasize that the update order for the coordinates is variable. Another variant of the Richardson algorithm is obtained by rewriting $Ax = b$ as $x = x - B(Ax - b)$ where B is any invertible matrix. Then the iterates are given by

$$x(t+1) = x(t) - b\left[Ax(t) - b\right].$$

The Gauss-Seidel variant of this update is also possible.

11.1 Convergence of the Classical Iterative methods

We will now prove a general theorem that encompasses the convergence of the classical iterative methods.

Theorem 11.1 *Suppose $b \in \mathbb{R}^n$ and $A = M - N \in \mathbb{R}^{n\times n}$ is nonsingular. If M is nonsingular and the spectral radius of M satisfies $\rho(M^{-1}N) < 1$, then the iterates $x^{(k)}$ defines by $x^{(k+1)} = M^{-1}(Nx^{(k)} + b)$ converge to $x = A^{-1}b$ for any starting vector $x^{(0)}$. Additionally, $\left\|e^{(k)}\right\| \leq \rho(M^{-1}N)^k \left\|e^{(0)}\right\|$. Consequently, we need $\log\left(\frac{\rho(M^{-1}N)}{e^{(0)}}\right)$ iterations to get an ϵ-approximate solution.*

Proof: *Let $e^{(k)} = x^{(k)} - x$ denotes the error in the kth iterate. Since $Mx = Nx + b$, it follows that $M(x^{(k+1)} - x) = N(x^{(k)} - x)$, and thus, the error in $x(k+1)$ is given by*

$$e^{(k+1)} = M^{-1}Ne^{(k)} = \left(M^{-1}N\right)^{k+1} e^{(0)}.$$

So, $\left\|e^{(k)}\right\| \leq \rho(M^{-1}N)^k \left\|e^{(0)}\right\|$. Taking the log of both sides are rearranging gives the result. ■

For the Jacobi algorithm, M is a diagonal matrix with the same diagonal entries as A, and N is the negative of the off-diagonal entries of A. For the Gauss-Seidel algorithm, M is the diagonal, and subdiagonal entries of A, while N is the negative of all of the entries above the diagonal.

12 Parallel Implementation of Classical Iterative Methods

We now discuss how to parallelize the previously introduced iterative methods. The Jacobi and Jacobi overrelaxation algorithms are easily parallelized. The update for each component can be computed completely independently of each other. Suppose that the ith processor has access the ith row of A. Then the update can be calculated via the inner product between two vectors, and after each component is updated, processor i passes $x_i(t)$ to all of the other processors. Another implementation is one in which the ith processor has access to the ith column of A. After each update, processor i passes $a_{ji}x_i(t)$ to processor j. Each component update is computed by summing up all of the received values.

In contrast, the Gauss-Seidel and successive overrelaxation algorithms are not well-suited to parallel implementation. Unfortunately, the modifications made to boost convergence introduced an inherently sequential component to their implementation. As written, the compute $x_i(t+1)$, processor i needs to know the value of $x_j(t+1)$ for all $j < i$. Fortunately, when A is sparse (as is often the case when A arises from the discretization of a partial differential equation), we can use a coloring scheme to parallelize the updates.

12.1 Coloring

In the Gauss-Seidel algorithm, one might hope that if each coordinate update does not **directly** affect all of the other coordinates then they can be implemented in parallel. This is exactly the

case. Precisely, notice that if $a_{ik} = 0$ for some $k < i$ then the updates

$$x_i(t+1) = -\frac{1}{a_{ii}} \left(\sum_{j<i} a_{ij} x_j(t+1) + \sum_{j>i} a_{ij} x_j(t) - b_i \right)$$

$$x_i(t+1) = -\frac{1}{a_{ii}} \left(\sum_{j<i, j\neq k} a_{ij} x_j(t+1) + \sum_{j>i} a_{ij} x_j(t) + a_{ik} x_k(t) - b_i \right)$$

are equivalent. We make this intuition precise via the following graph-theoretic notion of which Gauss-Seidel updates can be computed in parallel.

Given a dependency graph, $G = (V, E)$ representing the Gauss-Seidel iteration, a K-coloring is a map $C : V \to [K]$ that assigns a color $C(i)$ to each node in the graph.

Theorem 12.1 *There exists an ordering of the variables such that the Gauss-Seidel algorithm can be performed in K parallel steps if and only if there exists a K-coloring of the dependency graph where no positive cycle is entirely the same color.*

Proof: ($\Rightarrow$) *Consider an ordering of the variables with which the Gauss-Seidel iteration takes K parallel steps. We define our coloring map by $C(i) = k$ if node i is updated at the kth step. Then given any positive cycle $i_1, \ldots, i_m$, let i_l be the node that is updated first out of all $i_1, \ldots, i_m$. Since $(i_l, i_{l+1}) \in E$ this means $x_{i_{l+1}}$ depends on x_{i_l}, and the two variables cannot be updated in the same step. Consequently, they cannot be assigned the same color, and the cycle $i_1, \ldots, i_m$ has more than one color.*

($\Leftarrow$) *We will use the result that if a graph is a DAG, there there exists an ordering on the nodes such that if (i, j) is an edge, then node j comes before node i in the ordering.*

Now assume that there exists a K-coloring of $G = (V, E)$ such that no positive cycle is entirely the same color. We define the subgraphs G_k of G by only keeping nodes of the color k and the edges joining them. Then, by assumption, each of the subgraphs G_k is a DAG, so by the previous result, there is a topological ordering of each of these subgraphs. Then we order the nodes of G in increasing color order, where the ordering of nodes of the same color depends on the topological ordering of their associated subgraph. Consider the Gauss-Seidel update according to this ordering. Let nodes i and j have the same color k. Then if $(i, j) \in E$ and $(j, i) \in E$, then clearly x_i and x_j cannot be updated in parallel. It is not possible for $(i, j) \in E$ and $(j, i) \in E$ if nodes i and j have the same color k since each G_k is acyclic. If $(i, j) \in E$ and $(j, i) \notin E$, then node j is updated before node i, and so the computation of $x_j(t+1)$ only requires the value of $x_i(t)$ and not $x_i(t+1)$. The case where $(i, j) \notin E$ and $(j, i) \in E$ follows similarly. Therefore, every node of the same color can be updated in parallel, proving the result. ■

It's important to note that the dependency graph for the Gauss-Seidel updates is not necessarily a

DAG. To illustrate this, we consider an example where

$$
\begin{aligned}
x_1(t+1) &= f_1(x_1(t), x_3(t)) \\
x_2(t+1) &= f_2(x_1(t), x_2(t)) \\
x_3(t+1) &= f_3(x_2(t), x_3(t), x_4(t)) \\
x_4(t+1) &= f_4(x_2(t), x_4(t))
\end{aligned}
$$

Figure 12.1 depicts the dependency graph for this problem.

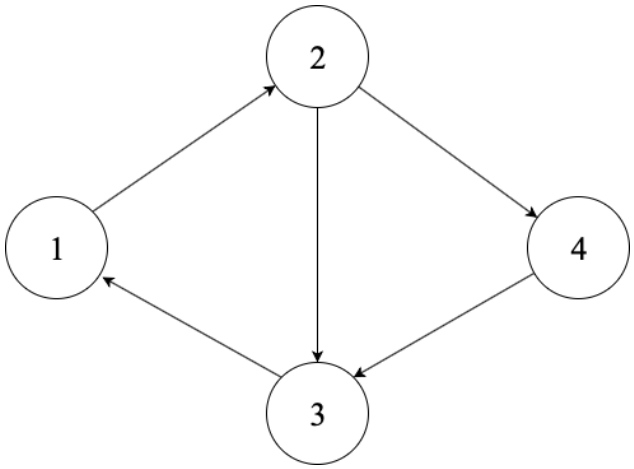
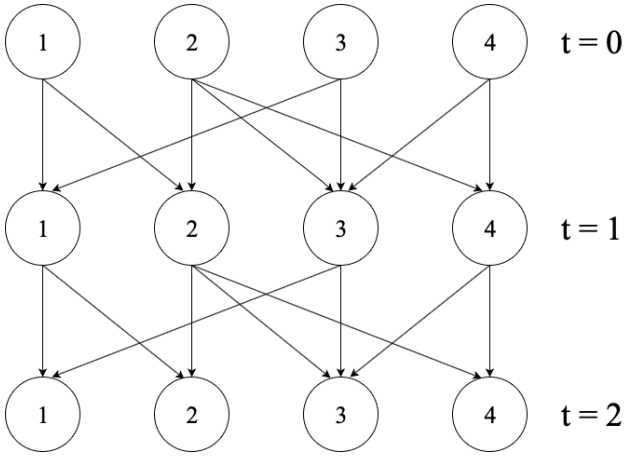


Figure 12.1 shows the DAG representing the Jacobi updates for the first two times steps.



Figures 12.1 and 12.1 show the DAGs representing the Gauss-Seidel updates for two different update orders. In figure 12.1, the variables are updated in order $x_1 \to x_2 \to x_3 \to x_4$, whereas in figure 12.1 the variables are updated in order $x_1 \to x_3 \to x_4 \to x_2$. The first ordering requires 3 parallel steps while the second only requires 2.

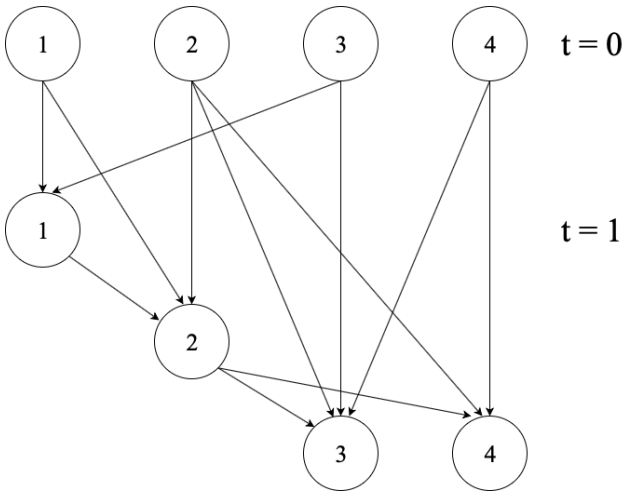


Figure 12: Gauss-Seidel updates with ordering $x_1 \to x_2 \to x_3 \to x_4$

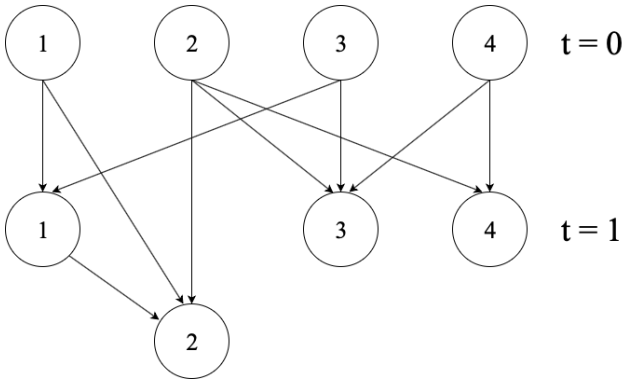


Figure 13: Gauss-Seidel updates with ordering $x_1 \to x_3 \to x_4 \to x_2$

13 Unconstrained Optimization

We will now show that the iterative algorithms for solving linear systems can be generalized to develop optimization algorithms.

In unconstrained optimization, the objective is to minimize some function $F : \mathbb{R}^n \to \mathbb{R}$. If F is continuously differentiable, and x^* is a minimizer of F, then $\nabla F(x^*) = 0$. Consequently, minimizing F is closely related to solving the nonlinear system of equations $\nabla F(x^*) = 0$. In general, without the appropriate guarantees of convexity, $\nabla F(x) = 0$ does not guarantee that x is a global minimizer of F. However, assuring that a critical point is a global minimizer is often intractable so many algorithms will settle for a critical point instead.

We will motivate the derivation of these algorithms by noticing that solving $Ax = b$ is equivalent to minimizing $F(x) = \frac{1}{2}x^T Ax - x^T b$. Then the gradient and Hessian of F are given by $\nabla F(x) = Ax - b$ and $\nabla^2 F(x) = A$. Then, the following algorithms can be interpeted as generalizations of the Jacobi overrelaxation, successive overrelaxation, and Richardson's algorithm respectively.

Jacobi Algorithm

$$x(t+1) = x(t) - \gamma[diag(\nabla^2 F(x(t)))]^{-1}\nabla F(x(t))$$

where $diag(\nabla^2 F(x(t)))$ is the diagonal matrix with the same diagonal elements as $\nabla^2 F(x(t))$

Gauss Seidel Algorithm

$$x_i(t+1) = x_(t) - \gamma \frac{\nabla_i F(\hat{x}(i,t)}{\nabla^2_{ii} F(\hat{x}(i,t)}$$

where $\hat{x}(i,t) = (x_1(t+1), \ldots, x_{i-1}(t+1), x_i(t), \ldots, x_n(t))$ is the most recent update of x

Gradient Algorithm Richardson's method can similarly be generalized as

$$x(t+1) = x(t) - \gamma F(x(t)).$$

And the Gauss-Seidel variant of the gradient algorithm is given by

$$x_i(t+1) = x_i(t) - \gamma \nabla_i F(\hat{x}(i,t)), \quad i = 1, \ldots, n$$

For a fixed $x \in \mathbb{R}^n$ where $\nabla F(x) \neq 0$, if $v \in \mathbb{R}^n$ is a vector such that $v^T \nabla F(x) < 0$ then v is said to be a descent direction. This is because since F is continuously differentiable, there exists a positive γ small enough such that $F(x + \gamma v) < F(x)$. Any algorithm that computes the next iterate by updating $x(t)$ along a descent direction is said to be a *descent algorithm*. The Jacobi, Gauss-Seidel, and gradient algorithms are all descent algorithms. They are generalized by the following scaled gradient algorithm

Scaled Gradient Algorithms

$$x(t+1) = x(t) - \gamma(D(t))^{-1}\nabla F(x(t))$$

where $D(t)$ is a scaling matrix. In practice D is often chosen to be diagonal so its inverse is just the diagonal matrix with entries $\frac{1}{D_{ii}}$. Not only is its inverse easy to calculate, but it is also more straightforward to implement in parallel.

13.1 Nonlinear Algorithms

The algorithms presented up until this point are called *linear algorithms* because each of the updates is a linear function of $\nabla F(x)$. *Nonlinear* or *coordinate descent* algorithms work by fixing all components of x but x_i and minimizing $F(x)$ with respect to only x_i.

In the nonlinear Jacobi algorithm, the minimizations with respect to each of the x_i are carried out simultaneously. Explicitly, the updates are given by

$$x_i(t+1) = \arg\min_{x_i} F(x_1(t), \ldots, x_{i-1}(t), x_i, x_{i+1}(t), \ldots x_n(t)).$$

Similarly, the Gauss-Seidel updates are carried out by using the most recent information as it is computed. Explicitly, the Gauss-Seidel updates are given by

$$x_i(t+1) = \arg\min_{x_i} F(x_1(t+1), \ldots, x_{i-1}(t+1), x_i, x_{i+1}(t), \ldots x_n(t)).$$

13.2 Parallel Implementation

Just as before, the parallel implementation of the Jacobi, and gradient algorithms are straightforward. We can assign the computation of $x_i(t)$ to the ith processor. After each $x_i(t)$ is computed, its value is communicated only to the processors which require it. In particular, the ith processor needs to know the current value of $x_j(t)$ if $\nabla_i F$ or $\nabla^2_{ii} F$ depends on x_j. For many problems arising in practice, both $\nabla_i F$ and $\nabla^2_{ii} F$ are sparse; this can be leveraged to drastically reduce the communication requirement of the algorithms.

In general, the Gauss-Seidel algorithms are unsuitable for parallel implementation except when the dependence graph is sparse. Then, the same coloring scheme technique can be applied to parallelize the computation.

One might recall that a common technique to boost convergence of optimization algorithms is to vary the step-size $\gamma(t)$ with each iteration such that $F(x(t) - \gamma \nabla F(x(t)))$ is minimized. Unfortunately, this minimization step is not amenable to parallelization.

14 Constrained Optimization

We will now consider the problem of constrained optimization. Here, our objective is to

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & F(x) \\ \text{subject to} & x \in \chi \end{array} \tag{11}$$

We will assume $F : \mathbb{R}^n \to \mathbb{R}$ to be continuously differentiable, and χ to be nonempty, closed, and convex. The necessary condition for $x \in \chi$ to be optimal is that $(y - x)^T \nabla F(x) \geq 0$ for all $y \in \chi$. If F is convex over χ then this condition is also sufficient.

14.1 Projected Gradient Descent

We will want to apply the descent algorithms of section 13 for constrained optimization. However, we will not be able to apply them directly. This is because even if an iterate is feasible i.e., $x(t) \in \chi$, it is not guaranteed that $x(t+1)$ is feasible. We will remedy this by simply projecting $x(t+1)$ back into the feasible set χ. In particular, we will define the projection operation as

$$\Pi_\chi(x) = \arg\min_{z \in \chi} \|x - z\|$$

and the iterates in the projected gradient algorithm are given by

$$x(t+1) = \Pi_\chi \left(x(t) - \gamma \nabla F(x(t)) \right)$$

Now we just need to show that our projection step is well-defined and the progress we have made in our descent step is not completely undone by the projection step. This is will follow from the projection theorem:

Theorem 14.1 (Projection Theorem)

1. *For every $x \in \mathbb{R}^n$ there exists a unique $z \in \chi$ that minimizes $\|x - z\|$ over all $z \in \chi$ (denoted as $\Pi_\chi(x)$).*
2. *For some $x \in \mathbb{R}^n$, $Z \in \chi$ is equal to $\Pi_\chi(x)$ is and only if $(y - z)^T (x - z) \leq 0$ for all $y \in \chi$.*
3. *The mapping $\Pi_\chi : \mathbb{R}^n \to \chi$ is continuous and nonexpansive. In other words,*

$$\|\Pi_\chi(x) - \Pi_\chi(y)\|_2 \leq \|x - y\|_2$$

for all $x, y \in \mathbb{R}^n$.

14.2 Parallel Implementation

The gradient projection algorithm is generally not well-suite to parallelization due to the projection step. One case where it is, though, is when the feasible set χ can be represented as the Cartesian product of constraint sets for the individual components of x i.e., $\chi = \prod_{i=1}^n [l_i, u_i]$. In this case, the projection onto χ is straightforward—if processor i is responsible for computing x_i, then it can simply project x_i onto $[l_i, u_i]$. This approach can also be generalized when the constraint set is a Cartesian product of sets i.e., $\chi = \prod_{i=1}^m \chi_i$. By viewing $\mathbb{R}^n$ as the Cartesian product of spaces $\mathbb{R}^{n_i}$, where $n_1 + \ldots + n_m = n$ and each χ_i is a closed convex subset of $\mathbb{R}^{n_i}$, it is apparent that projecting x onto χ is equivalent to projecting the appropriate components of x onto χ_i individually. In other words $\Pi_\chi(x) = \prod_{i=1}^m \Pi_{chi_i}(x^{(i)})$, where here we let $x^{(i)}$ denote the appropriate components of x corresponding to χ_i. For example, if $\chi = \{x \in \mathbb{R}^4 | x_1 + x_2 = 0,\ x_3 + x_4 = 0\}$, then we can express $\chi = \{x \in \mathbb{R}^2 | x_1 + x_2 = 0\} \otimes \{x \in \mathbb{R}^2 | x_3 + x_4 = 0\}$, and $x^{(1)} = (x_1, x_2)$, $x^{(2)} = (x_3, x_4)$.

A similar discussion also applies to the parallel implementations of the scaled gradient algorithm. In general, we cannot expect to be able to compute $x(t+1) = x(t) - \gamma (D(t))^{-1} \nabla F(x(t))$ in a

distributed manner. However, if $(D(t))^{-1}$ has a "nice" structure, it is possible. We briefly discussed the case where $D(t)$ is diagonal, and thus $(D(t))^{-1}$ is diagonal and each components updates can be computed in parallel. In general, if there is a permutation, represented by matrix P, where $P(D(t))^{-1}$ is block diagonal, then the components in the same block (after permutation) need to be updated together, but those that are not in the same block can be updated independently of each other.

14.3 Distributed Nonlinear Algorithms

If we assume χ to be a Cartesian product, then it makes sense for us to consider the projected nonlinear Jacobi and Gauss-Seidel algorithms. They operate simply by restricting the the minimization to the feasible sets for each component. The updates are given as follows:

Jacobi

$$x_i(t+1) = \arg \min_{x_i \in [l_i, u_i]} F(x_1(t), \ldots, x_{i-1}(t), x_i, x_{i+1}(t), \ldots x_n(t)).$$

Gauss-Seidel

$$x_i(t+1) = \arg \min_{x_i \in [l_i, u_i]} F(x_1(t+1), \ldots, x_{i-1}(t+1), x_i, x_{i+1}(t), \ldots x_n(t)).$$

14.4 Parallelization by Decomposition

Previously we showed that a number of descent algorithms can be parallelized. However, these methods are not always applicable, especially for constrained optimization (which require the constraint set to be the Cartesian product of constraints on individual components). We will now explore how specific problem structure can be exploited to derive a parallel solution algorithm by find a suitable transformation of the problem. Oftentimes the dual optimization problem is much more amenable to parallel implementation than the original problem. We will illustrate this idea with several examples.

14.5 Quadratic Programming

The objective of a quadratic programming problem is to solve a problem of the form

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & \frac{1}{2} x^T Q x - b^T x \\ \text{subject to} & Ax \leq c \end{array} \tag{12}$$

where $Q \in \mathbb{R}^{n \times n}$ is positive definite, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^n$, and $c \in \mathbb{R}^m$ are all known. The dual of this problem is given by

$$\begin{array}{ll} \underset{u \in \mathbb{R}^m}{\text{minimize}} & \frac{1}{2} u^T (AQ^{-1}A^T) u + (c - AQ^{-1}b)^T u \\ \text{subject to} & u \geq 0 \end{array} \tag{13}$$

If u^* is the optimal solution to problem (13), then the optimal solution x^* can be recovered by the relation $x^* = Q^{-1}(b - A^T u^*)$. Notice that while the constraint set defined by $\{x|Ax \leq c\}$ cannot generally be expressed as the Cartesian product of simpler sets, $u \geq 0$ certainly can, making the methods we have previously discussed suitable for solving the dual problem. In particular, we will consider the non-linear Jacobi algorithm.

Let $D(u) = \frac{1}{2}u^T Pu + r^T u$, with $P = AQ^{-1}A^T$,and $r = c - AQ^{-1}b$ be the dual objective function. Note that $\nabla_j D(u) = r_j + \sum_{k=1}^m p_{jk}u_k$. Because the dual objective is convex, its minimizer can be found by finding u such that $\nabla D(u) = 0$. Using the expression we derived for $\nabla_j D(u)$, we see that

$$\arg\min_{u_j} = \tilde{u}_j = -\frac{\gamma}{p_{jj}}\left(r_j + \sum_{k \neq j} p_{jk}u_k\right)$$

Then, taking into account the nonnegativity constraint, $u_j = \max\{0, \tilde{u}_j\}$. Our iterates are then given by

$$u_j(t+1) = \max\left\{0, u_j(t) - \frac{\gamma}{p_{jj}}\left(r_j + \sum_{k=1}^m p_{jk}u_k(t)\right)\right\}.$$

Notice that each of components can be updated in parallel.

14.6 Separable Strictly Convex Programming

Suppose that the space $\mathbb{R}^n$ is represented as the Cartesian product of spaces $\mathbb{R}^{n_i}$, $i = 1, \ldots, m$ where $n_1 + \ldots + n_m = n$, and consider the problem

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & \sum_{i=1}^m F_i(x^{(i)}) \\ \text{subject to} \quad & e_j^T x = s_j, \quad j = 1, \ldots, r, \\ & x^{(i)} \in P_i, \quad i = 1, \ldots, m \end{aligned} \tag{14}$$

Where $F_i : \mathbb{R}^{n_i} \to \mathbb{R}$ are strictly convex, $x^{(i)}$ are the appropriate components of x, e_j are given vectors in $\mathbb{R}^n$, s_j ar es alres, and P_i are bounded polyhedral subsets of $\mathbb{R}^{n_i}$. Then, the dual problem is given by

$$\underset{p \in \mathbb{R}^r}{\text{maximize}} \quad q(p) \tag{15}$$

where,

$$q(p) = \min_{x^{(i)} \in \mathbb{R}^{n_i}} \left(\sum_{i=1}^m F_i(x^{(i)}) + \sum_{j=1}^n p_j(e_j^T x - s_j)\right) \tag{16}$$

This is now separable

$$q_i(p) = \min_{x_i \in P_i} \left(F_i(x^{(i)}) + \sum_{j=1}^r p_j \hat{e}_{ji}^T x_i\right) \tag{17}$$

where $\hat{e}_{ji}$ denotes the appropriate components of e_j corresponding to $x^{(i)}$. The evaluation of the dual function is amenable to parallelization with each separate processor computing a component $q_i(p)$ of $q(p)$.

15 Introduction to Optimization for Machine Learning

We will now shift our focus to unconstrained problems with a separable objective function, which is one of the most prevalent setting for problems in machine learning. Formally stated, we wish to solve the following problem:

$$\underset{w}{\text{minimize}} \quad F(w) := \sum_{i=1}^{n} F_i(w, x_i, y_i) \tag{18}$$

where we can interpret x_i's as the input data, the y_i's as the output data, and w as the parameter we wish to optimize over. The F_i's are objective functions ("loss functions") that are designed for the problem at hand. A few commonly used examples include:

- **Least squares regression** Here the hypothesis is that the output data y_i is explained by the input data x_i by a linear equation and some noise. Specifically, we postulate that $y_i = +\epsilon_i$. The squared error loss function is given by $F_i(w, x_i, y_i) = ||w^T x_i + b - y_i||_2^2 = \|\epsilon_i\|_2^2$
- **Support vector machine (SVM)** Here the outputs mark each of the x_i's as coming from one of two categories, -1 or 1, and the goal is to build a model that assigns new examples to one of the two categories. If the goal is "maximum-margin" classification, the hinge loss is appropriate. The hinge loss is given by $F_i(w, x_i, y_i) = \max\{0, 1 - y_i(w^T x_i)\}$. When $w^T x_i$ and y_i have opposite signs (meaning incorrect classification), the loss increases linearly with $|w^T x_i|$. When the have the same sign (meaning correct classification) and $|w^T x_i| \geq 1$ the loss is zero, but if $|w^T x_i| < 1$ a loss is still incurred because the point is not classified with enough margin (hence why it is good for "maximum-margin" classification). (convex)

In this setting, the optimization problem has some aspects that are suited for distributed computing, such as regularization and hyperparameter tuning, but these are quite straightforward and not particular interesting from an algorithmic or distributed design perspective. The two quantities we are interested in with regards to the scaling such algorithms are:

- **Data parallelism**: How does the algorithm scale with n (number of training points)?
- **Model parallelism**: How does the algorithm scale with d (number of parameters)?

Gradient descent, and stochastic gradient descent are some of the more widely used methods for solving this optimization problem. In this lecture, we will first prove the convergence rate of gradient descent (in the serial setting); the number of iterations needed to reach a desired error tolerance will inherently determine the total depth of the (stochastic) gradient descent algorithm. We will then discuss how gradient descent and stochastic gradient descent trade-off work per iteration and number of iterations needed for convergence. This section will conclude with HOGWILD!, an algorithm that exploits sparsity to circumvent the inherently sequential aspect of computing iterations.

15.1 Gradient Descent

The general idea of gradient descent is based on initializing w at random and performing the sequential updates:

$$w_{k+1} \leftarrow w_k - \alpha \nabla f(w_k),$$

where the convergence depends on α and the structural properties of F itself.

15.2 Convergence of Gradient Descent

In general, function minimization is impossible unless we make some sort of structural assumption about the function itself (imagine a discontinuous function that jumps around arbitrarily at each point!). Fortunately, for most practice problems it's reasonable to impose some assumptions about the structure of the problem, namely convexity[40] and smoothness[41]. Theorems 15.3 and 15.7 state the convergence rates of gradient descent on L-smooth and L-smooth, μ-strongly convex functions respectively. This section will be dedicated to building up the necessary background for proving these theorems.

Definition 15.1 (L-smooth) *A differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be L-smooth is for all $x, y \in \mathbb{R}^n$, we have that*

$$\|\nabla f(x) - \nabla f(y)\|_2 \leq L \|x - y\|$$

The gradient of a functions measures how the function changes when we move in a particular direction from a point. If the gradient were to change arbitrarily quickly, the old gradient does not give us much information at all even if we take a small step. In contrast, smoothness assures us that the gradient cannot change too quickly—therefore we have an assurance that the gradient information is informative within a region around where it is taken. The implication is that we can decrease the function's value by moving the direction opposite of the gradient.

Lemma 15.1 *Let $f : \mathbb{R}^n \to \mathbb{R}$ be differentiable, then for all $x, y \in \mathbb{R}^n$ and $x_t = x + t(x - y)$ for $t \in [0, 1]$. Then,*

$$f(x) - f(y) = \int_0^1 \nabla f(x_t)^T (y - x) dt$$

And so,

$$f(x) - f(y) - \nabla f(x)^T (y - x) = \int_0^1 (\nabla f(x_t) - \nabla f(x))^T (y - x) dt$$

Proof: *Let $g(t) := f(x + t(x - y))$. Then, because $g'(t) = \nabla f(x_t)^T (y - x)$, the first claim follows from the fundamental theorem of calculus. The second claim follows from the first by subtracting $\nabla f(x)^T (y - x)$ from both sides of the equation.* ■

[40] Although most problems in machine learning are not convex, convex functions are among the easiest to minimize, making their study interesting

[41] We can also often forgo the smoothness assumption by using subgradients instead of gradients. We will assume smoothness for illustrative purposes, because extensions to the nonsmooth case are straightfoward

Lemma 15.2 *If $f : \mathbb{R}^n \to \mathbb{R}$ be L-smooth. Then for all $x, y \in \mathbb{R}^n$ we have that*

$$|f(y) - (f(x) + \nabla f(x)^T(y-x))| \leq \frac{L}{2} \|x-y\|_2^2$$

Proof: *Let $x_t = x + t(x-y)$, then we can apply the previous lemma to get*

$$\begin{aligned}
|f(y) - (f(x) + \nabla f(x)^T(y-x))| &\leq |\int_0^1 (\nabla f(x_t) - \nabla f(x))^T (y-x) dt| \\
&\leq \int_0^1 |(\nabla f(x_t) - \nabla f(x))^T (y-x)| dt \\
&\leq \int_0^1 \|\nabla f(x_t) - \nabla f(x)\| \, \|y-x\| \, dt
\end{aligned}$$

where the second and thrid inequality follow from Cauchy Schwartz.
We can now use the definition of smoothness to get a bound on $\|\nabla f(x_t) - \nabla f(x)\|$. Note that,

$$x_t - x = x + t(x-y) - x = t(x-y).$$

So, by smoothness, $\|\nabla f(x_t) - \nabla f(x)\| \leq t \|x-y\|$. Plugging this back into the integral,

$$\begin{aligned}
|f(y) - (f(x) + \nabla f(x)^T(y-x))| &\leq \int_0^1 \|\nabla f(x_t) - \nabla f(x)\| \, \|y-x\| \, dt \\
&\leq L \|x-y\|_2^2 \int_0^1 t dt \\
&= \frac{L}{2} \|x-y\|_2^2
\end{aligned}$$

yields the result. ■

We can now analyse the convergence of gradient descent on L-smooth functions.

Theorem 15.3 *Gradient descent on L-smooth functions, with a fixed step-size of $\frac{1}{L}$ achieve an ϵ-critical point in $\frac{2L(f(x_0)-f_*)}{\epsilon^2}$ iterations.*
Proof: *Applying Lemma 15.1, we get*

$$|f(w_{k+1}) - (f(w_k) + \alpha \|\nabla f(w_k)\|_2^2)| \leq \frac{\alpha^2 L}{2} \|\nabla f(w_k)\|_2^2$$

So if we pick $\alpha = \frac{1}{L}$, then $f(w_{k+1}) \leq f(w_k) - \frac{1}{2L} \|\nabla f(w_k)\|_2^2$. So, in k gradient descent steps, we have an iterate w_k such that

$$f_* - f(w_0) \leq f(w_k) - f(w_0) \leq \frac{1}{2L} \sum_{i=0}^{k-1} \|\nabla f(w_i)\|_2^2 .$$

Rearranging, we get

$$\frac{1}{k} \sum_{i=0}^{k-1} \|\nabla f(w_i)\|_2^2 \leq \frac{2L(f(w_0) - f_*)}{k}$$

So for one of the w_i's, $\|\nabla f(w_i)\|_2^2 \leq \frac{2L(f(w_0)-f_*)}{k}$. So, if we want to find an ϵ-critical point i.e., a point where $\|\nabla f(w)\|_2 \leq \epsilon$, we need at least $\frac{2L(f(x_0)-f_*)}{\epsilon^2}$ iterations. ■

Remember in the previous lecture we mentioned that guarantees that we are close to a global optimum are hard to come by? This is one of those cases, so we need to settled for an ϵ-critical point instead.

One of the assumptions that will allow us to obtain global optimality guarantees is convexity.

Definition 15.2 (μ-strongly convex) *A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be μ-strongly convex for $\mu \geq 0$ if for all $x, y \in \mathbb{R}^n$ and $t \in [0, 1]$ we have that*

$$f(ty - (1-t)x) \leq tf(y) + (1-t)f(x) - \frac{\mu}{2}t(1-t)\|x-y\|_2^2$$

A function is said to be *convex* if the above relation holds for $\mu = 0$, and typically *strongly convex* implies that $\mu > 0$. We will first state a couple helper lemmas that will be will be useful in analyzing gradient descent on μ-strongly convex functions (their proofs are included in the appendix for the interested reader).

Lemma 15.4 *1. A differentiable function is μ-strongly convex if and only for all $x, y \in \mathbb{R}^2$,*

$$f(y) \geq f(x) + \nabla f(x)^T(y-x) + \frac{\mu}{2}\|x-y\|_2^2$$

2. A twice differentiable function f is μ-strongly convex if and only if for all $x \in \mathbb{R}^n$

$$z^T\nabla^2 f(x)z \geq \mu\|z\|_2^2$$

Proof: *Suppose that for all $x, y \in \mathbb{R}^2$,*

$$f(y) \geq f(x) + \nabla f(x)^T(y-x) + \frac{\mu}{2}\|x-y\|_2^2$$

Then, for a given $x, y \in \mathbb{R}^2$, let $x_t = x + t(y-x)$ where $t \in [0, 1]$. Then,

$$\begin{aligned} f(y) &\geq f(x_t) + \nabla f(x_t)^T(y-x_t) + \frac{\mu}{2}\|y-x_t\|_2^2 \\ &= f(x_t) + (1-t)\nabla f(x_t)^T(y-x) + \frac{\mu}{2}(1-t)^2\|y-x\|_2^2 \end{aligned}$$

Similarly,

$$\begin{aligned} f(x) &\geq f(x_t) + \nabla f(x_t)^T(x-x_t) + \frac{\mu}{2}\|x-x_t\|_2^2 \\ &= f(x_t) - t\nabla f(x_t)^T(y-x) + \frac{\mu}{2}t^2\|y-x\|_2^2 \end{aligned}$$

Taking t times the first expression and $1-t$ times the second expression, and combining the two yields,

$$f(ty - (1-t)x) \leq tf(y) + (1-t)f(x) - \frac{\mu}{2}t(1-t)\|x-y\|_2^2.$$

Now we assume

$$f(ty-(1-t)x) \leq tf(y)+(1-t)f(x)-\frac{\mu}{2}t(1-t)\left\|x-y\right\|_2^2$$

to be true. Rearranging,

$$\begin{aligned} tf(y) \geq f(ty-(1-t)x) &\leq -(1-t)f(x)+\frac{\mu}{2}t(1-t)\left\|x-y\right\|_2^2 \\ f(y) &\geq \frac{f(ty-(1-t)x) \leq -(1-t)f(x)+\frac{\mu}{2}t(1-t)\left\|x-y\right\|_2^2}{t} \\ &= f(x)+\frac{\mu}{2}(1-t)\left\|y-x\right\|_2^2+\frac{f(x_t)-f(x)}{t} \end{aligned}$$

Since $\frac{f(x_t)-f(x)}{t}$ *converges to* $\nabla f(x)(y-x)$ *as* $t \to 0$*. Taking the limit of expression above as* $t \to 0$ *yields,*

$$f(y) \geq f(x)+\nabla f(x)^T(y-x)+\frac{\mu}{2}\left\|x-y\right\|_2^2$$

Assume that for all $x \in \mathbb{R}^n$

$$z^T\nabla^2 f(x)z \geq \mu\left\|z\right\|_2^2$$

Then, for $x, y \in \mathbb{R}^n$ *let* $x_\alpha = x+\alpha(y-x)$ *with* $\alpha \in [0,1]$*,*

$$\begin{aligned} f(y) &= f(x)+\nabla f(x)^T(y-x)+\int_0^1\int_0^t (y-x)^T\nabla^2 f(x_\alpha)(y-x)d\alpha dt \\ &\geq f(x)+\nabla f(x)^T(y-x)+\int_0^1\int_0^t \mu\left\|y-x\right\|_2^2 d\alpha dt \\ &\geq f(x)+\nabla f(x)^T(y-x)+\frac{\mu}{2}\left\|y-x\right\|_2^2 \end{aligned}$$

We can apply the previous lemma to get the result.
Now assume that f *is* μ*-strongly convex, and for* $x, t \in \mathbb{R}^n$*, define* $x_t = x+tz$ *for all* $t \in \mathbb{R}$*. Let* $g(t) = f(x_t)$*. Then* $g'(t) = \nabla f(x_t)^T z$ *and* $g''(t) = z^T\nabla^2 f(x_t)z$*. Furthermore,*

$$g''(0) = \lim_{t\to 0}\frac{g'(t)-g'(0)}{t} = \lim_{t\to 0}\frac{(\nabla f(x_t)-\nabla f(x))^T z}{t} = \lim_{t\to 0}\frac{(\nabla f(x_t)-\nabla f(x))^T(x_t-x)}{t^2}$$

Applying the previous lemma, we have that

$$f(x_t) \geq f(x)+\nabla f(x)^T(x_t-x)+\frac{\mu}{2}\left\|x_t-x\right\|_2^2$$

and

$$f(x) \geq f(x_t)+\nabla f(x_t)^T(x-x_t)+\frac{\mu}{2}\left\|x_t-x\right\|_2^2$$

By definition, $x_t - x = tz$*, so, adding these together yields*

$$(\nabla f(x_t)-\nabla f(x))^T(x_t-x) \geq \mu t^2\left\|z\right\|_2^2,$$

proving our result. ■

Ultimately, convexity will allow us to translate gradient information into distance from the optimum and suboptimality of the function value. Previously, we were able to find an ϵ-critical point using gradient descent only assuming L-smoothness. This ends up not being very informative because when a function is non-convex, its gradients can equal zero at a saddle point or a maximum—these points are certainly not optimal. However, this cannot be the case for convex functions. In particular, if we define $X_*(f) := \{x | f(x) = f_*\}$ to be the set of minimizers of f, then $x_* \in X_*(f)$ iff and only if $\nabla f(x_*) = 0$.

Lemma 15.5 *If $f : \mathbb{R}^n \to \mathbb{R}$ is L-smooth, then for all $x_* \in X_*(f)$ and $x_* \in \mathbb{R}^n$,*

$$\frac{1}{2L} \|\nabla f(x)\|_2^2 \leq f(x) - f_* \leq \frac{L}{2} \|x - x_*\|_2^2$$

Proof: *We will use the inequality,*

$$f(y) \leq f(x) + \nabla f(x)^T (y - x) + \frac{L}{2} \|y - x\|_2^2$$

for all $x, y \in \mathbb{R}^n$. Consider x, y where $y = x - \frac{1}{L}\nabla f(x)$. Then, from our previous analysis we have that $f(y) \leq f(x) - \frac{1}{2L} \|\nabla f(x)\|_2^2$. By definition of optimality, $f_* \leq f(y)$. So,

$$f_* \leq f(y) \leq f(x) - \frac{1}{2L} \|\nabla f(x)\|_2^2$$

and we have the left hand side of the inequality, $\frac{1}{2L} \|\nabla f(x)\|_2^2 \leq f(x) - f_*$.

Now, we plug in x for y, and x_* for x to get,

$$f(x) \leq f(x_*) + \nabla f(x_*)^T (x - x_*) + \frac{L}{2} \|x - x_*\|_2^2 .$$

Since $\nabla f(x_*) = 0$, this inequality becomes

$$f(x) \leq f(x_*) + \frac{L}{2} \|x - x_*\|_2^2 .$$

Thus, proving the right side of the inequality. ■

We can prove similar bounds if f is μ-strongly convex.

Lemma 15.6 *If $f : \mathbb{R}^n \to \mathbb{R}$ is smooth and μ-strongly convex for $\mu > 0$, then for all $x_* \in X_*(f)$ and $x_* \in \mathbb{R}^n$,*

$$\frac{1}{2\mu} \|\nabla f(x)\|_2^2 \geq f(x) - f_* \geq \frac{\mu}{2} \|x - x_*\|_2^2$$

Proof: *The proof follows analogously to the L-smooth case. Here we use the relation*

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{\mu}{2} \|x - y\|_2^2$$

Plugging in x for y, and x_* for x we get,

$$f(x) \geq f(x_*) + \nabla f(x_*)^T (x - x_*) + \frac{\mu}{2} \|x_* - x\|_2^2$$

Using the fact that $\nabla f(x_*) = 0$ yields,

$$f(x) \geq f(x_*) + \frac{\mu}{2} \|x_* - x\|_2^2$$

which is the right side of the inequality. To prove the left side, note that

$$f(x_*) \geq \min_y f(y) + \nabla f(x)^T (y - x) + \frac{\mu}{2} \|x - y\|_2^2 \geq f(x) - \frac{1}{2\mu} \|\nabla f(x)\|_2^2$$

since $y = x - \frac{1}{\mu} \|\nabla\| f(x)$ minimizes the middle expression. ■

Now that we can relate gradient information to suboptimality and distance from an optimum, we can determine the convergence rate of gradient descent for strongly convex functions.

Theorem 15.7 (Strongly Convex Gradient Descent) *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a L- smooth, μ-strongly convex function for $\mu > 0$. Then for $x_0 \in \mathbb{R}^n$ let $x_{k+1} = x_k - \frac{1}{L} \nabla f(x_k)$ for all $k \geq 0$. Then,*

$$f(x_k) - f_* \leq \left(1 - \frac{\mu}{L}\right)^k (f(x_0) - f_*).$$

Consequently, we require $\frac{L}{\mu} \log\left(\frac{f(x_0) - f_*}{\epsilon}\right)$ iterations to find an ϵ-optimal point.

Proof: *We have previously shown that $f(x_{k+1}) \leq f(x_k) - \frac{1}{2L} \|\nabla f(x_k)\|_2^2$. From Lemma 15.6, we know that $\|\nabla f(x_k)\|_2^2 \geq 2\mu(f(x_k) - f_*)$. Putting this all together,*

$$\begin{aligned} f(x_{k+1}) - f_* &\leq f(x_k) - f_* - \frac{1}{2L} \|\nabla f(x_k)\|_2^2 \\ &\leq f(x_k) - f_* - \frac{\mu}{L}(f(x_k) - f_*) \\ &= (1 - \frac{\mu}{L})(f(x_k) - f_*) \end{aligned}$$

So, applying this bound repeatedly gives us

$$f(x_k) - f_* \leq \left(1 - \frac{\mu}{L}\right)^k (f(x_0) - f_*).$$

Using the fact that $1 + x \leq e^x$, and picking $k \geq \frac{L}{\mu} \log\left(\frac{f(x_0) - f_*}{\epsilon}\right)$ gives the convergence rate. ■

We now have convergence rates for gradient descent. The next important step is understanding the complexity of computing the gradient, and with that some of the challenges of parallelizing gradient descent. We will use linear least squares as an illustrative example.

15.3 Analyzing least squares gradient descent

The linear least squares objective is given by

$$F(w) = \sum_{i=1}^n F_i(w, x_i, y_i) = \sum_{i=1}^n ||x_i^T w - y_i||_2^2.$$

This has work $O(nd)$ and depth $O(\log n + \log d)$. This object function has gradient

$$\nabla_w F = \sum_{i=1}^{n} \nabla_w F_i(w) = \sum_{i=1}^{n} 2(x_i^T w - y_i)x_i \quad \in R^d$$

Here, we see that the gradient is just a re-scaling of x_i. Computing the gradient for a single datapoint can be done in $O(\log d)$ depth (the computation being dominated by the dot product operation). The global gradient $\nabla_w F$ then requires $O(\log d) + O(\log n)$ depth. The previous analysis shows that to achieve ϵ error, gradient descent requires $\mathcal{O}(\log \frac{1}{\epsilon})$ iterations. Thus total depth for gradient descent is $\mathcal{O}(\log \frac{1}{\epsilon}(\log n + \log d))$, which means that our algorithm is slow even with multiple processors. This is because each of the iterations is inherently sequential, and thus not easily parallelized.

15.4 Stochastic Gradient Descent

Stochastic gradient descent (SGD) reduces the work of gradient descent by "sampling" the gradient. Rather than computing all of $\nabla F = \sum_{i=1}^{n} \nabla F_i(w, x_i, y_i)$, at each iteration, we randomly pick some subset of the $F_i(w, x_i, y_i)$ and compute their gradients. Why does this work? If we sample uniformly at random, in expectation, the sampled gradients will equal the true gradients. Let s_k be index uniformly sampled at time step k (we use one training point for illustrative purposes. However, you can sample batches of gradients as well). Then, the SGD iteration as follows:

$$w_{k+1} \leftarrow w_k - \alpha \nabla F_{s_k}(w_k).$$

We have reduced the work per iterations by n, which is a drastic reduction if our training set is very large (as they tend to be). However, this comes at the cost of a slower convergence rate (in terms of the number of iterations) than gradient descent.

Theorem 15.8 *If there exists a constant G such that $\mathbb{E}[\|\nabla f_i(x)\|^2] \leq G^2$ and $f(x)$ is μ-strongly convex. Then, with step-sizes $\gamma_k = \frac{1}{\mu k}$, we have*

$$\mathbb{E}[\|x_k - x_*\|^2] \leq \frac{\max\{\|x_1 - x_*\|^2, \frac{G^2}{\mu^2}\}}{k}.$$

This means that SGD achieves ϵ error after $\mathcal{O}(\frac{1}{\epsilon})$ iterations.
Proof: *The proof will be given as an exercise in the homework, and has been omitted from these notes.* ■

Notice the $\mathcal{O}(\cdot)$ notation hides constants such as the Lipschitz constant, depending on F, which can be pretty large.

The table below summarizes the performance of gradient descent vs stochastic gradient descent[42] where ϵ is our tolerance for convergence. (To get this table, we have assumed F_i has a reasonable Lipschitz constant, and that evaluating each F_i requires $O(d)$ work and $\log(d)$ depth.)

[42]There's a hidden $O(\log(n))$ cost to sampling from a uniform random variable

	# iterations	work/iter	depth/iter	total work	total depth
GD	$O(\log \frac{1}{\epsilon})$	$O(nd)$	$O(\log d + \log n)$	$O(nd \log \frac{1}{\epsilon})$	$O((\log \frac{1}{\epsilon})(\log d + \log n))$
SGD	$O(\frac{1}{\epsilon})$	$O(d)$	$O(\log d)$	$O(\frac{d}{\epsilon})$	$O(\frac{\log d}{\epsilon})$

In the case of Gradient Descent, the number of iterations is trapped in a log expression, so total depth is as well (this is desirable). However, if we want an error of $\epsilon = 10^{-6}$, this becomes problematic in SGD, which requires many more iterations than GD. On the other hand, the total work for SGD does not depend on n, which is powerful if we are working with massive data sets. Because the iterations of SGD are inherently serial, they generally cannot be parallelized. However, when problems are sufficiently sparse[43] we can parallelize these updates.

We now consider a more refined statement of our problem:

$$\underset{w}{\text{minimize}} \quad F(w) := \sum_{i=1}^{n} F_i(w_e, x_i, y_i) \tag{19}$$

where $e \subseteq [d]$ denotes a subset of $\mathbb{R}^d$ and w_e represents the appropriate components of w. We now just have to manage conflicts on F_i's that share the same components of w.

Managing conflicts is tricky because CPU's do not have direct access to the memory; instead computations can only be done on CPU cache. Data is read from the memory to CPU cache, and after the computations are finished, results are pushed back to memory. Therefore, a processor has no way of knowing whether other processors have local updates that have not yet been pushed to memory.

Conflicts are typically dealt with with a "coordinate" or "memory lock" approach. An exclusive lock essentially tells other processors that one of the processors is currently updating a variable and prevents them from making changes until it is done. This can provide some speedup over serial SGD, and it is straightforward to prove correctness of the algorithm. However load imbalances result in idle-time of the processors, rendering the algorithm inefficient.

15.5 Hogwild!

HOGWILD! [8] works by simply removing the locks. Processors have access to shared memory, and can potentially overwrite each others' work. This works if the problem is sufficiently sparse.

Many machine learning problems are sparse: for example matrix factorization, matrix completion, graph cuts, graph or text classification etc. Generally, this means if multiple processors try to write to w at the same time, "most" of the time they will be writing to different components of w. The proof of HOGWILD! then relies of proving that the instances where multiple processors try write to the same components of w will not completely derail the convergence of SGD.

The main idea/insight in the proof is that HOGWILD! can be interpreted as a noisy serial SGD. The noise comes from the instances where multiple processors try write to the same components of w, and can be bounded using the sparsity of the "conflict graph", which represents instances where updates might try to write to the same component.

[43] Although, the convergence HOGWILD! has only been proven for sparse problems, it has seems to work well for dense problems as well, and we do not really know why

Theorem 15.9 *Let F be μ-strongly convex, with $\|\nabla F_i\|^2 \leq M^2$. If the number of samples that overlap in time with a single sample during the execution of HOGWILD! is bounded as*

$$\tau = \mathcal{O}\left(\min\left\{\frac{n}{\overline{\Delta}_C}, \frac{M^2}{\epsilon\mu^2}\right\}\right)$$

HOGWILD!, with stepsize $\gamma = \frac{\epsilon\mu}{2M^2}$ reaches an accuracy of $\mathbb{E}[\|x_k - x_*\|^2] \leq \epsilon$ after

$$T \geq \mathcal{O}(1)\frac{M^2 \log\left(\frac{a_0}{\epsilon}\right)}{\epsilon\mu^2}$$

iterations. Where $\overline{\Delta}_C$ is the average degree in the conflict graph.

Algorithm 14: HOGWILD!

```
Initialize w in shared memory // in parallel, do
for i = {1, ..., p} do
    while TRUE do
        if stopping criterion met then
            break
        end
        Sample j from 1, ..., n uniformly at random.
        Compute f_j(w) and ∇f_j(w) using whatever w is currently available.
        Let e_j denote non-zero indices of x_i
        for k ∈ e_j do
            w_(k) ← w_(k) − α [∇F_j(w)]_(k)
        end
    end
end
```

Communication in GD/SGD happens at each iteration: $O(\frac{1}{\varepsilon})$ vs $O(\log \frac{1}{\varepsilon})$. Thus large differences in the cost of communication between a single machine with shared memory and distributed cluster may change the economics of the choice between GD and SGD. In practice, we see single machine architectures using SGD on big beefy GPUs for things like deep learning whereas we may see distributed systems running GD.

16 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.

16.1 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[44] can take 10 million nanoseconds to fetch the same data, i.e. 100k× slower.

[44] not a solid state disk

- 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.

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.

17 Communication Networks

We will model the communication structure of a distributed system as a network of processors of connected by communication links. Specifically, in a network of n processors can be modeled as a graph $G = (V, E)$ where $V = [n]$ represents the processors and there is an edge $(i, j) \in E$ if processor i can send a message to processor j. Each processor has its own local memory and computational capabilities, and shares intermediate results with other processors through groups of bits (called *packets* sent over the communication links. The communication graph G may be directed or undirected depending on whether the links are uni-directional or bi-directional. Moreover, each link is characterized by its **latency** and **bandwidth**. Latency is defined as the time it takes to for data to travel from one point to another. Bandwidth is defines as the rate at which data can be transferred over a link during a fixed amount of time. Ultimately, these limits on communication will fundamentally limit the efficiency of our distributed algorithms.

There are a number of causes of delay in communications:

1. *Communication Processing Time* is defined as the time needed to prepare a message for transmission. Before transmitting, the information needs to be assembled into packets with addressing and control information appended to each packet, the link on which to transmit each packet needs to be selected, and the packets need moved to the appropriate buffers.

2. *Queueing Time* is defined as the time a packet waits in the queue before it is transmitted. This can happen for a number of reasons: for example, the link is being used to send other packets, or the allocation of a link to several contending packages is being decided, or the transmission is delayed to ensure the appropriate resources i.e., buffer space is available at the destination.

3. *Transmission Time* is defined as the time required to transmit all of the bits in the packet.

4. *Propagation Time* is defined as the time between the transmission of the last bit of a packet and depends on the physical distance that data must travel and the medium it travels through. For example, in a fiber optic cable, data travels at the speed of light.

We won't deal with the specifics of networking, but instead will lump together our communication costs in terms of latency and bandwidth.

We will now revisit the Jacobi iterative method for solving systems of linear equations to illustrate the central role of the communication network on algorithm efficiency. Recall, that if we are trying to solve $Ax = b$ (where we assume $A \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^n$, $b \in \mathbb{R}^n$ and we meet the appropriate conditions for convergence of the Jacobi iteration), the Jacobi iterates are given by

$$x_i(t+1) = -\frac{1}{a_{ii}} \left(\sum_{j \neq i} a_{ij} x_j(t) - b_i \right).$$

Suppose we have a linear array of processors p processors. Specifically, we assume there is a bi-directional link between processors i and $i + 1$. Figure 17 illustrates a linear array of processors.

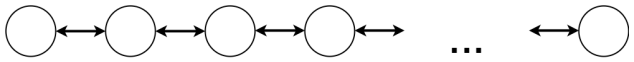


Figure 14: Linear Array of Processors

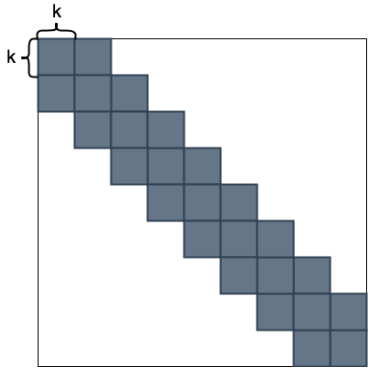


Figure 15: Block tri-diagonal structure

Consider the scenario where each processor is assigned to the computation of a consecutive block of k elements of x (i.e., processor 1 calculates $x_1, \ldots, x_k$, processor 2 calculates $x_{k+1}, \ldots, x_{2k}$ etc.). If A is block tri-diagonal with blocks of size k, (Figure 17 illustrates such an A), then the communication complexity is low relative to the computation complexity. Specifically, at each iteration processor i needs the values of variables calculated by processors $i-1$ and $i+1$. Consider a communication protocol where at each iteration each processor i passes its components of x to first $i+1$ and second $i-1$. Note that processors will never try to use the same communication link. Therefore, the total communication complexity per iteration is $\Theta(k)$ (where hides $\Theta(\cdot)$ the latency and bandwidth constants). The total computational complexity carried out by each processors is $\Theta(k^2)$.

Now if A has a banded structure with bands k elements apart, (Figure 17 illustrates such an A), the computation and communication complexities are on the same order. Specifically, at each iteration processor i needs the values of variables calculated by all other processors. Moreover, since all of the processors are connected along a single path, transmissions between the processors will need to use the same communications links. Consider a communication protocol where for each iteration, all processors send their components of x first to processor 1, then to processor 2, etc. The total communication complexity per iteration is $\Theta(kp)$, and the total computation complexity per iteration is $\Theta(kp)$ as well. A smarter way to distribute this computation is if processor 1 was assigned components $1, k+1, 2k+1, \ldots$, processor 2 was assigned components $2, k+2, 2k+2, \ldots$ and so on. In this case, no communication needs to happen between iterations, and so the total communication complexity per iteration is 0, while the total computation complexity per iterations is still $\Theta(kp)$.

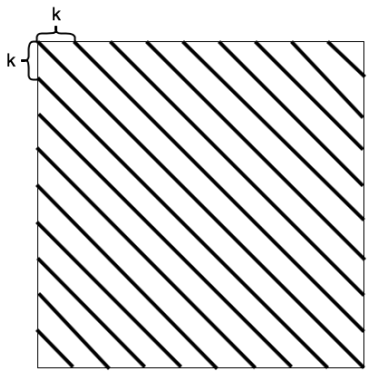


Figure 16: Banded structure

18 Cluster Computing, Broadcast Networks, and Communication Patterns

Commodity cluster computing refers to using a large number of low-cost, low-performance commodity computers, connected via ethernet, working in parallel instead of using fewer high-performance and high-cost computers. The ethernet link between processors means that every processor can communicate with every other processor in the cluster. In other words, the communication network is a complete graph. This doesn't mean that all processors should always be passing messages to each other, though, as this could easily overload the bandwidth of the communication links. Consequently, communication will need to be limited to ensure that the bandwidth constraints are respected. One could theoretically design communication patterns that are tailored to the problem at hand, however, the overhead of fault tolerance makes this unreasonable. Multiple programs must be written for fault tolerance: one program for failures, one for scheduling and putting code where it needs to be, and another for data locality. MapReduce and Spark will automatically handle this overhead, however, this comes at the cost of full control of the communication patterns, which are typically restricted to one of three patterns: (1) All to One, (2) One to All, and (3) All to All. We will now analyze the cost of communication for each of these patterns.

18.0.1 All to one communication with a *driver* machine

Consider a scenario where computation is distributed among multiple machines and the results are sent to a single *driver* machine, as shown in Fig. 17. If all machines are directly connected to the *driver* machine, the bottleneck of this communication is the network interface of the *driver* machine. Let p be the number of machines (excluding the *driver*), L be the latency between each pair of machines (specifically, L is the time it takes for the first message from a machine to arrive at the driver machine), and B be the bandwidth of the network interface of the *driver* machine. If all machines send a message of size M to the *driver* and *driver*'s network interface is saturated by every single message (meaning that the driver can only take in messages from other machines one at a time), then each single message sent takes time proportional to

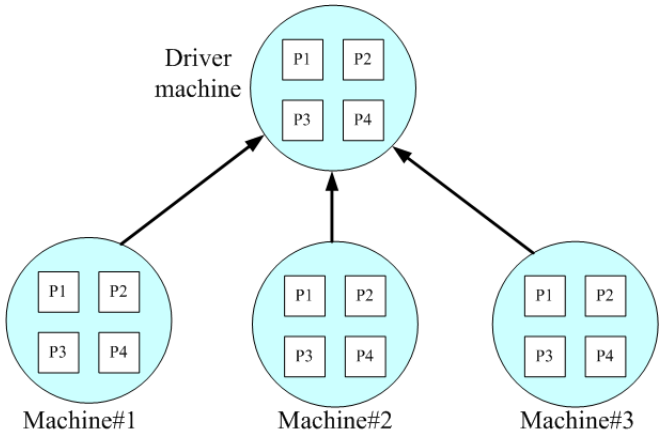


Figure 17: All to one communication with *driver* machine

$$L + \frac{M}{B}$$

Thus the overall communication time scales with:

$$p\left(L + \frac{M}{B}\right)$$

18.0.2 All to one communication as *Bittorent Aggregate*

Another algorithm for all to one communication is known as *Bittorent Aggregate*. Just as before, we assume that each machine carries out a portion of the computation, and results need to be aggregated at one machine. Instead of performing the aggregation at once, it is done incrementally aggregating the results held by pairs of machines. The aggregation pattern can be seen as a tree structure or as depicted in Fig.18. Assume we are aggregating results from p machines, the results can be aggregated to a single machine in $\log_2 p$ rounds. Let L be the latency and B be the bandwidth between each pair of machines and in each aggregation round a message of size M is sent between machine pairs. The total communication time is:

$$(\log_2 p)\left(L + \frac{M}{B}\right)$$

18.0.3 One to All communication

We can think of one to all communication as all to one communication with data flow in the opposite direction. Suppose the *driver* machine needs to sends messages of size M to p other machines. The *driver* machine's network interface is still the bottleneck of communication. Following the an analogous computation, communication cost is the same as one to all communication with *driver* machine.

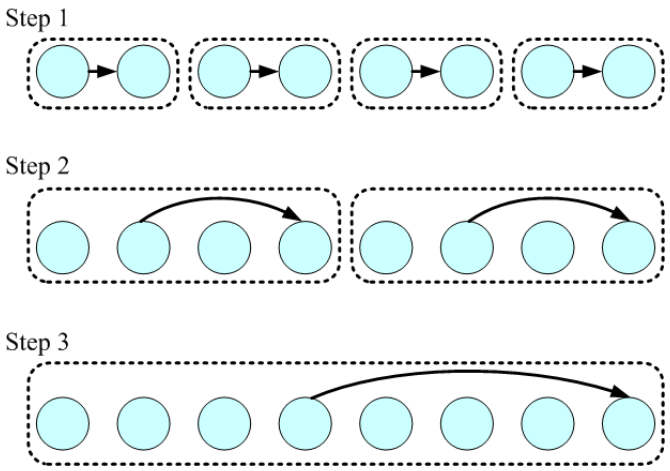


Figure 18: All to one communication with *Bittorent Aggregate*

Similarly, we can use the concept of *Bittorent Aggregate* for one to all communication as well. In this case, the message is relayed among machines in a tree structure. Then the message can be spread among all machines within $O(\log p)$ rounds.

18.0.4 All to all communication

While we have previously stated that all to all communication can easily saturate bandwidth constraints, there are some operations where all machines have to communicate with each other.

For example, sorting requires all to all communication. Suppose we are trying to sort a large set of integers which exceeds the storage of a single machine. If the numbers are shuffled and distributed among multiple machines, without communication with the other machines, no individual machine can know which number are stored on the other machines. Therefore, we require all to all communication for each machine to determine how the numbers relate to the numbers stored in other machines.

Other examples of problems that require all to all communication are JOIN and GROUPBY. In fact, sorting is often a subroutine in implementing these two operations.

19 Optimization, scaling, and gradient descent in Spark

We will now consider two different optimization algorithms: gradient descent, and stochastic gradient descent. In this class, we'll implement gradient descent in Spark, and then see why SGD not suitable for Spark. Then we'll learn how to implementSGD regardless of the data-flow paradigm of the language we're computing in. For the remainder of this section, we will assume we have a separable objective functions, of the form

$$\min_x \sum_{i=1}^{n} F_i(x)$$

where $w \in \mathbb{R}^d$ and F_i is the loss-function applied to a training point i. We won't make any assumption about the specific form of each F_i.

The main question we are interested in is how our algorithm scales both in n, the number of data points we have, and d the dimension of our model. In the first part of the lecture, we'll assume n is too large to fit on a single machine but that d numbers can fit on a single machine's RAM. Recall that gradient descent starts with a random initial vector, x_0, say initialized to all zeros. Then at each iteration $x_{k+1} \leftarrow x_k - \alpha \sum_{i=1}^{n} \nabla F_i(x_k)$, where $\nabla F_i(\cdot)$ is a vector itself.

19.0.1 Implementing Gradient Descent in Spark

The following block of code demonstrates an implementation of gradient descent in Spark:

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
For (i <- 1 to numIterations):
  val gradient = points.map(p => $\nabla F_p(w)$) .reduce(_+_)
  w -= alpha * gradient
```

The first line reads in a text-file of training points and their associated labels. The operation, `parsePoint` is a closure which takes as argument a single like of the text-file of training points and their associated labels, and outputs a clean d-dimensional x_i vector and a single label y_i. After parsing the files, we cache the RDD with the training points. The action `cache` tells Spark that we're going to use a particular RDD frequently, so each machine should try to hold that its part of the RDD in memory as much as possible rather than flushing data to disk. Since we need to access this RDD every iteration of gradient descent, we will want to keep it in memory. Recall that `cache` is an action (as opposed to a transformation), so Spark actually kicks off computation and reads in the text file, applying the `parsePoint` closure to each part of the distributed file.[45] The next line initializes the model parameters to a d-dimensional vector of all zeros. Finally, each iteration of the for loop implements an iteration of gradient descent. The closure

$$\texttt{points.map(p =>} \nabla F_p(w)\texttt{)}$$

maps each of the training points to their associated gradients of the loss function. Specifically, $\nabla F_p(w)$ is the gradient of F_p evaluated at point w. For example, if $F_i(w) = (w^T x_i - y_i)^2$, then ∇F_i may be calculated symbolically, and we may plug this formula into $\nabla F_p(\cdot)$. Then we calculate the full gradient as the sum of the individual gradients—we use reduce for this. Recall that Spark uses lazy evaluations, the the computation is not kicked off until this reduce step. Under the hood, Spark takes into account communication cost between machines, scheduling, and executing operations locally in a PRAM model.

[45] Suppose we have a 1 terabyte dataset, and we have a small amount of memory, say 10 gigabytes. Suppose we wish to process the large dataset. We can read directly off the large terabyte slowly, or we can read directly off the ram for faster access. The question becomes which part of the data set to store in ram since we can't fit the entire object in memory. We commonly employ Least Recently Used caching, which evicts items from ram according to which piece of data was least recently used. This is a simple heuristic, but works in practice.

19.0.2 Broadcasting in Spark

Recall that by default, when a `map` happens, anything that is needed for the `map` to happen is shipped out to the workers. So in the line

$$\texttt{points.map(p => } \nabla F_p(w)\texttt{).reduce(\_+\_)}$$

since the function that maps each training point to it gradient depends on w, a copy of w will be sent to **every** CPU of **every** machine. This happens, because by default, we assume w is being modified. Consequently, we must store w separately to avoid any concurrent write issues. However, when we calculate each gradient step, w is not being modified so there is no chance of a concurrent write. We are wasting storage by not storing w in shared memory. To get around this, we can broadcast `w` so that it only gets sent to each machine a single time. To do this, we define `w` as a broadcast variable. The following block of code illustrates gradient descent with the model parameters as a broadcast variable.

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
var w_br = sc.broadcast(w)
For (i <- 1 to numIterations):
  val gradient = points.map(p => $\nabla F_p$(w_br.value)) .reduce(_+_)
  w -= alpha * gradient
  w_br = sc.broadcast(w)
End
```

When we initialize w, we first declare it as a broadcast variable. In each of the iterations of gradient descent, we need to refer to the *value* of w. Finally, we re-broadcast w after it gets updated. In addition to being more storage efficient on each machine, i.e. storing a single copy of the data object instead of multiple, we also take advantage of bit-torrent broadcasting when we use Spark's broadcasting.

19.1 Analysis

Recall, gradient descent requires $\mathcal{O}(\log \frac{1}{\epsilon})$ iterations to achieve an ϵ-optimal solution if f is L-smooth, and μ-strongly convex. While the computation of the gradients can be distributed across the cluster of machines, the iterations of gradient descent are inherently serial. At each iteration, Spark implements a synchronization barrier to ensure that w is fully updated before the workers begin computing the next iteration. Just as in the parallel case, if there are stragglers, the other workers are idle waiting for the next iteration. The depth of the computation of typically is the bottleneck rather than the gradient updates.

Communication time The map is embarrassingly parallel, and requires no communication. The broadcast requires one to all communication, and the reduce requires all-to-one communication. So, per iteration, our total communication time scales with

$$2\log_2(p)(L + \frac{m}{B})$$

[46] Technically, when we call `points.map()`, we serialize code and send it to each machine. One reason we chose Scala to implement Spark is that it's easy to serialize code. We remark that there is a function called `allReduce` which merges the concept of a reduce and broadcast. After a reduce, the result of the reduce ends up on the driver machine. After an `allReduce` the result of the `reduce` gets sent to all machines via a broadcast; the broadcast happens simultaneously with the regular reduce.

20 Distributed Summation

Suppose we have data sharded between p machines and we simply want to sum up all the data. If we model each machine with our PRAM model, we can use the parallel summation algorithm discussed in the first half of the course. Note that these parallel summations can happen concurrently at all the machines. Once the individual machines are done, suppose their results are communicated to a master and aggregated there. If we assume that each message has size m with L latency and B bandwidth, then this requires $p(L + \frac{m}{B})$ time.[47]

What's wrong with this approach? If every machine tries to send its partial sums to the master, then its network card will be the bottle-neck. The other machines are capable of aggregating the partial sums as well. Moreover, the network cards on all other machines lay idle. This means we are not efficiently using our computation and communication resources. A more efficient algorithm for distributed summations by using *bit-torrent* broadcasting, and extrapolating our parallel summation algorithm. Specifically, on the first time step machines m_1, m_2 sum up their results, machines m_3, m_4 sum up their results, etc. From each pair, we select exactly one machine to propagate the result to the next level in our tree. This allows us to take full advantage of our compute resources. Moreover, by using as many links in the network as possible, this allows us to avoid bottlenecks at any one network card.

This algorithm results in $\log_2(p)$ rounds of communication before termination. Each round takes $L + m/B$ communication time, for a total of $\log_2(p)\,(L + m/B)$ communication time.

Notice that the choice to compute partial sums between pairs of machines is arbitrary. We could instead sum up between multiple machines to use as much of the bandwidth as possible. In practice, we often will not worry about this $\log p$ factor—this is because machines are costly, so it's very rare to so many machines that the $\log p$ term is the limiting factor.

After all rounds of communication take place, only one machine has the sum. If all the machines need to use this sum, we need to send it to them via a one-to-all broadcast. We can use **bit-torrent broadcasting** to accomplish this.

21 Simple random sampling

On a single machine, if we have n data points and want to sample an observation uniformly at random, it suffices to first generate a random number between 0 and $n-1$, and select the data point with this index. In the distributed setting, we now have to deal with the challenge of incomplete knowledge. We will often face scenarios where we do not have a priori knowledge of the amount of data being fed into our algorithms. Moreover, this data may come from various sources such as the disk, or network.

For example, consider running an algorithm using data from the Twitter firehose[48]With giga-

[47] L is on the order of 1-2 milliseconds for a local network, or a couple hundred milliseconds is the data need to be sent across the world. Ultimately, we are limited by the speed of light; we can't transfer physical information or data any faster than that.

[48] Twitter firehose is the complete stream of public messages. For reference, Users send 500 million tweets every day. This corresponds to 6000 tweets per second.

bytes of data coming in per second, it is infeasible to store all the data and randomly sample from it. Additionally, with a constant stream of data, there is no notion of when the all of the data has come in.

The following algorithm gives a method to sample from a stream of data uniformly at random.

Algorithm 15: Sampling from a stream uniformly at random

```
Initialize return value r as empty, and stream counter k as 1.
while stream is not yet empty do
    Read an item from the stream, and flip a coin with probability 1/k.
    if coin comes up heads then
        r ← s
    end
    Increment k
end
return r
```

We will now prove the correctness of the streaming uniform sampling algorithm. Precisely, we need to show that after n items have been read, the value stored in r has equal probability $1/n$ of being any of these n items. We will prove the claim via induction. The base case follows by observing that when $n = 1$, our stream size will have counter value $k = 1$, and so r will be set to the first element in the stream with probability $1/k = 1$, hence r correctly represents a random sample from the singleton set containing only the first item in the stream.

For the inductive step, we assume that our claim holds for a stream up to length $n-1$. Consider the state of the algorithm after the nth item is read from the stream and processed. Our stream size counter has value $k = n$, and so r will be set to the newest element of the stream with probability $1/n$, and we leave r changed with probability $(n-1)/n$. We now need to show that r has equal probability of being any of the stream items seen so far. Specifically, we need to show that that $\Pr(r = s_i) = 1/n$ for all $i = 1, \ldots, n$, where s_i denotes the ith item of the stream.

We already proved that r has value s_n with probability $1/n$, i.e. $\Pr(r = s_n) = 1/n$. Now we need to consider $\Pr(r = s_i)$ for $i < n$.

If r was not replaced on the most recent step, it had the same value as it had after $n-1$ steps. By the inductive hypothesis, this value of r represents a sample uniform at random from the first $n-1$ items of the stream, i.e. $\Pr(r = s_i = \frac{1}{n-1}$ for $i = 1, \ldots, n-1$. Since r is *not replaced* with probability $(n-1)/n$, we see that

$$\Pr(r = s_i) = \frac{n-1}{n} \cdot \frac{1}{n-1} = \frac{1}{n}$$

for all $i = 1, \ldots, n-1$. Hence after n items have been read from the stream, we have $\Pr(r = s_n) = 1/n$ for every $i = 1, \ldots, n$. This completes the inductive step.

22 Distributed sort

With distributed summation, simply extending the parallel summation algorithm worked well. What if we tried to implement quicksort on a cluster?

Recall in quicksort, we select a pivot p uniformly at random. Then we construct the sets L and R where L contains all items less than p and R contains all items greater than p. We recursively call quicksort on L and R until all items have been sorted.

Now, suppose we simply try to implement this algorithm on a cluster. Specifically, assume we have B machines, were each machine has some portion of the data. We first need to randomly pick a pivot p. Each machine can sample uniformly at random from its data. From there, we can select one of theses B samples based on a weighted index of the amount of data on each machine.

Now that we have a pivot, we need to construct L and R. This is problematic because we are typically using distributed sorting because the input data is too large to fit on one machine. Consequently, L and R likely will not fit on one machine either. We could try to allocate half of the machines to storing L and half to R, and recursively using this approach. However, in the worst case, this could result in reshuffling all of the data in each recursive call. We might try to get around this by not shuffling data around maintaining $L_1, \ldots, L_M$ and $R_1, \ldots, R_M$ on each machine. The recursive call would involve picking one pivot out of $L_1, \ldots, L_M$ and other from $R_1, \ldots, R_M$, and sorting according to those pivots. Ultimately, this would result in sorted arrays on each machine. However, merging these sorted array would still take linear time in the number of inputs.

Often, recursion requires too much overhead from preparing and returning function inputs and outputs to be used in a distributed computing framework.

It is helpful to think about how our output might differ from the parallel setting. When our algorithm is done, we want the data on each individual machine to be sorted, and for there to be an ordering on the machines. For example, if we select $d_1, d_2, \ldots, d_B$ from machines $1, \ldots, B$ respectively, then we requires that $d_1 < d_2 < \ldots < d_B$. We also want the quantity of data on each of the machines to be balanced.

Suppose we knew the true distribution of our data. Then each machine can sort its own data and send its data from the "first histogram bin" to one machine, second to another and so on. Distributed sort works by estimating the true distribution of the data. Below, we give the algorithm for distributed sorting.

Algorithm 16: Distributed sort

1. Each machine sorts its own data
2. Each machine samples 100 data points from its local data and sends to a driver node
3. Driver approximates integer distribution by sorting its sampled data
4. From this estimate, driver can determine cutoff thresholds for each machine
5. All thresholds are broadcast to all machines, i.e. each machine knows its relevant data
6. All-to-all communication: each machine shuffles data to correct location
7. Each machine sorts the incoming (sorted) partitions that it gets sent in linear time

A note on determining cutoff points In steps 4 and 5, we specifically seek a sequence of cutoff points such that each bin of our histogram contains an equal number of data points, i.e. the total number of data values is divided evenly by the number of machines. Notice that these cutoff points may not be uniformly spaced if the distribution of data is not uniform. To get each cutoff point, once the driver has sorted the sampled data, we just need to look at every kth value in the sorted array, where k is a multiple of the length of the array over the number of machines; this gives an estimate of the cutoff point for the kth machine.

A note on all to all communication In step 6, because each machine has sorted its data (on disk), we can stream data from disk and send it across the network, where the receiving machine reads the data directly in to disk. This means we don't need to temporarily store the data in RAM, spend timing finding the relevant data on disk. Moreover, the driver machine broadcasts the cutoff points to all machines, so the data exchange in step 6 can be implemented in a peer to peer fashion. This can be done one of two ways. In the first, each machine pushes the appropriate portions of its data to each other machine. In the second, each machines queries every other machine for the appropriate portions of the data. The second is often preferable because fault-tolerance is simple. If a machine dies during this process, it can simply query the other machines for the data again. In contrast, with the first method, other machines cannot inherently detect that one of the machines has died, and thus do not know if they need to resend the data.

What's left? After the all-to-all communication, we still need to merge the results back to one node. But we've seen how to merge two sorted arrays in linear time on our homework.

23 Introduction to MapReduce

MapReduce is a programming paradigm that is particularly amenable to processing large data sets in a distributed fashion over a cluster of machines. The user defines a *map* function, and *reduce* function. The run-time system then handles the details of parallelization, fault-tolerance, data distribution, and load balancing. This is a powerful framework allows users to automatically parallelize and distributed large-scale computations.

23.1 Programming Model

Users of the MapReduce library write two scripts specifying a *map* function and *reduce* function.

The *map* function takes an input set of key/value pairs and outputs an intermediate set of key/value pairs. The MapReduce library guarantees that all values with the same key are grouped together before passing them to the *reduce* functions.

The *reduce* function takes as input **one** intermediate key and a set of values for that key, and outputs a (potentially smaller) set of values. Typically, each *reduce* invocation produces only one output value, but it is not required.

23.1.1 Examples

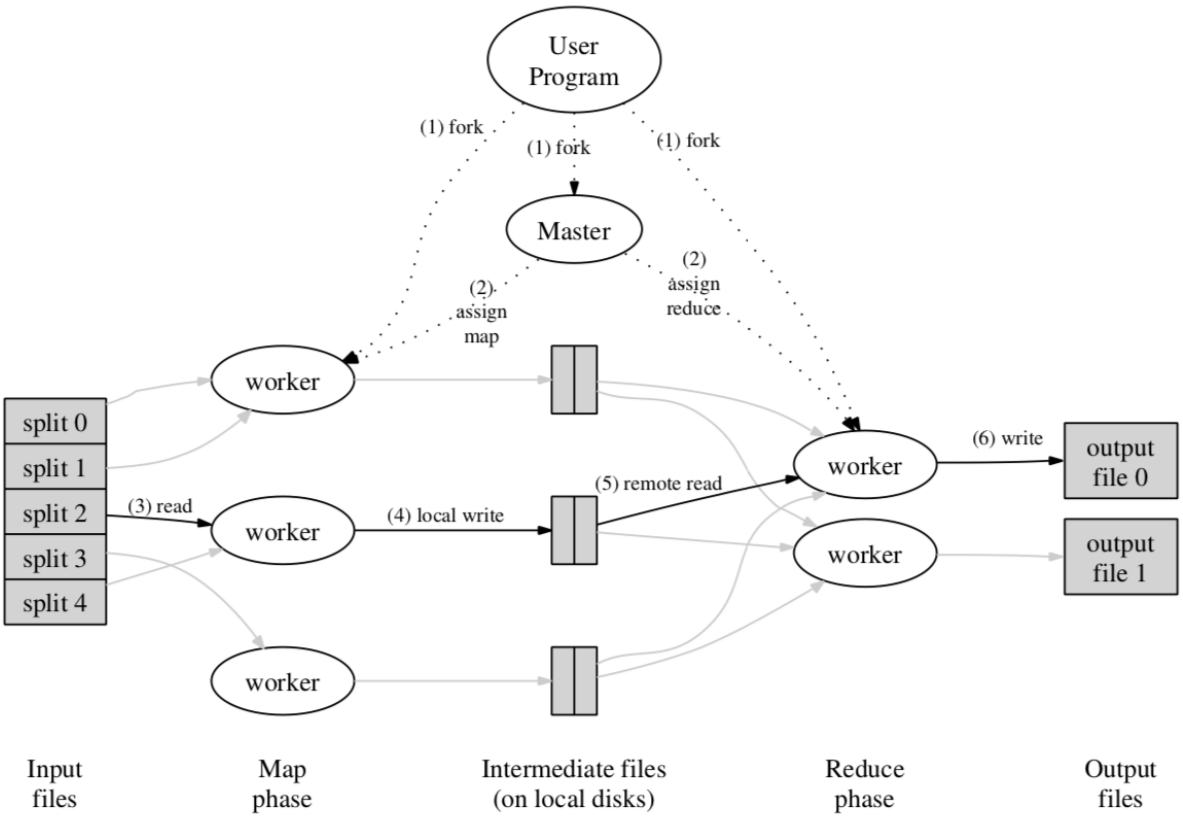
Counting Occurrences Suppose we want to count the number of occurrences of each word in a collection of documents. In this case, the input key/value pairs would be the document name and document contents respectively. The *map* function would output key/value pairs of the form $(word, 1)$ for each *word* in the documents. The reduce function would take an input a single word key and a list of counts for that word. For example, if the word "w" appears k times in all of the documents, then there will be k key/value pairs (w, 1). The *reduce* function would then iterate through the pairs for an individual word and add up the counters.

Inverted Index Suppose we want to locate all occurrences of a word in a set of documents. The map function would take as input document IDs and document contents, parse each document and output a sequence of (word, document ID) pairs. The reduce function takes the (word, document ID) pairs and outputs (word, list(document ID)).

23.2 MapReduce Execution

There are multiple possible implementations of the MapReduce interface, with the appropriate one depending on the computing environment. This section overviews the implementation targeting the computing environment at Google: a large cluster of commodity PCs connected together with switched Ethernet.

The *map* function is executed in parallel by multiple machines by partitioning the input data into M "splits"; the mapping function can be executed independently and in parallel on each of the splits. Similarly, the *reduce* function can be executed in parallel by partitioning the set of keys (output from the *map* function) into R splits. The partitioning function and number of splits, R, are specified by the user.



When a user calls the MapReduce function, the following sequence of steps are executed (depicted in Figure 23.2).

1. The MapReduce library in the user program partitions the input files into M splits (typically of 16 - 64 MBs per split).

2. Copies of the program are also started up on each machine in the cluster. One of the copies is the designated master, while the remaining are workers that are assigned tasks by the master. In total there are $M + R$ tasks that need to be assigned.

3. Workers that are assigned *map* tasks are each assigned a split of the input data, on which it runs the *map* function. The output key/value pairs are buffered in memory.

4. The buffered pairs are written to a one of R partitions of the local disk. The locations of these buffered pairs are passed back to the master program, which passes the locations to the *reduce* workers when they are assigned a *reduce* task.

5. When a *reduce* worked is assigned a *reduce* task, it receives a list of buffered locations from the master, and reads in all the intermediate data from these locations. It sorts the read-in data by the intermediate keys so that all key/value pairs with the same key are grouped

together—there is no guarantee that all of the data passed to a single *reduce* worker have the same key.

6. Each reduce worker iterates over the sorted data, and starts a new reduce task when a new key is encounter. The outputs of each of the reduce invocations is appended to a final output file for the partition.

7. Once all of the map and reduce tasks are complete, the output of the entire MapReduce execution is available as the aggregtion of the R output files. Typically, the outputs for the partitions are not combined.

During the execution of the MapReduce, the master coordinates computation across the worker nodes. Specifically, for each of the map and reduce tasks, the master stores a state ("idle", "in-progress", or "completed") and the ID of the worker assigned to the task if it is "in-progress" or "completed". When a map task is completed, the master stores the location and sizes of the R intermediate file regions produced by the map task. This information is passed onto the reduce worked with "in-progress" reduce tasks.

23.3 Fault Tolerance of MapReduce

One of the key features of the MapReduce paradigm is that it automatically builds in robustness to machine failure. When data is being processed by hundreds or thousands of machines, it is likely that one will fail, so the library must recover from failures efficiently. The master keeps tabs of worker failure by periodically pinging each worker. If it receives no response (within a set time period) it marks that worker as failed, and any map tasks tasks that were completed by that worker are marked as "idle" so they can be assigned to another worker. Similarly, "in-progress" map or reduce tasks are rest to "idle" if their workers have failed. Completed map tasks need to be re-executed because their output are stored on the local disk of the failed machine, and thus cannot be accessed. On the other hand, because the outputs of the reduce functions are stored in a global file system, they are still accessible even if their worker dies. Consequently, completed reduce tasks do not need to be reexecuted if their worker fails. When a map tasks is reassigned, the reduce workers are notified of the reassignment, so they can read data from the appropriate map workers.

MapReduce does not build in safe-guards in case of master failure. This is because the master is only one node, and the chance of one particular node failing are unlikely.

Outline

- Map reduce and indexing
- Sparse matrix multiplication using SQL
- Joins using map reduce

23.4 Map Reduce

Recall what map reduce does: it's a tool for putting pairs that have the same key together. The map reduce environment gave the following promise:

- Map phase: emits pairs of the form $\langle \text{key}, \text{value} \rangle$ (shuffle).
- Shuffle phase: places pairs with the same key on a single machine.
- Reduce phase: performs computations on pairs with the same key.

On the face of it, this may look trivial, however, this construct is powerful. If your write your code according to this contract you get fault tolerance, you get distribution, and utilization of all cores in the cluster.

Example Suppose we've indexed the web by crawling it with a thousand machines. So now each machine has stored on it a part of the web; each machine is full of HTML and pointers to other HTML pages. From this, we'd like to build a search engine kind of like Google. So, how can we apply map reduce to a search engine? Given a key word, we need to be able to *very* quickly figure out which web-pages have that key in them; this is the most basic search engine functionality. If we have an inverted-index, we can do this efficiently.

An inverted-index is a list of words, where each word is associated with the pages it occurs in. So, we'd like to get to a point where we have a list

$$\text{word}_i \to \{\text{page}_1, \text{page}_2, \ldots\}$$

How can we create such an index using map reduce? All we have is a jumble of web-pages, and we want a map between the word itself and the pages on which it occurs. We only have one hammer at our disposal, map-reduce, which puts pairs with the same key together. So, naturally, the key should be the word itself, and the value should be the url.

So, each mapper will examine the web-page that it has been given. For each word in the web-page, emit the word as the key and the url as the value.

The above code is low-level, and can be tedious to write ourselves. Instead, we move onto SQL, and specifically multiplying sparse matrices.

Algorithm 17: Mapper

```
   // Map phase
1  for w ∈ H do
2  |  emit (w, h_uid)
3  end
   // Reduce phase
4  Reduce(w, ⟨url_1, url_2, ...,⟩)
```

23.5 Sparse Matrix-Vector Multiply using SQL

First of all, let's figure out how our matrix is represented. Since our matrix M is sparse, we need the following representation

$$(i, j, \text{value})$$

i.e. for each non-zero entry we store the row and column indices alongside the corresponding value. A sparse vector x is represented as (i, value).

Now, imagine that our matrix is an SQL table, where the columns in the table is i, j, and the value; similarly for our matrix, represented as a table with a column for the i index and a column for the value.

How can we compute Mx using SQL operations?[49] We realize that if $Mx = b$, then each entry of b an inner product between corresponding row of M and vector x. But how can we prepare these dot-products? We first perform a join on M and x where we join M on its column index j and x on its element index i. Our resulting table looks like

$M.j(x_i)$	$M.i$	M.value	x.value

Notice that where there are duplicates in our join, a cartesian product will appear. I.e. for every single entry in matrix M, we will get the corresponding value from vector x.

M =

i	j	value
1	2	17.5
2	2	16.3

x =

i	value
1	2.71
2	3.14

The resulting table after our `inner-join`

[49] First attempt: We can first group by i on matrix table M; at this point, each entry in the resulting table will correspond to a row in M, where the value is a function of the values in the columns of M for said row. When we do a `group-by`, we need to specify what kind of aggregation function we perform to combine the values for a particular group. What kind of aggregation should we use? We abandon this approach.

$M.j(x_i)$	$M.i$	M.value	x.value
2	1	37.5	3.14
2	2	18.3	3.14

The second SQL statement simply adds a new column onto this table. Thew column will look like

$M.j(x_i)$	$M.i$	M.value	x.value	mult
2	1	37.5	3.14	37.5×3.14
2	2	18.3	3.14	...

We've done the multiplication in each dot-product. Now we need to perform the summation, i.e. perform a group by i and sum over our column `mult` which we added above. We'd like to take all of the multiplications that are associated with a particular dot-product and sum them up, so our aggregation function is a summation on `mult`. Notice that we only need as output the index i and corresponding value of Mx, so we can remove other columns.

Now that we've done this in SQL, how can we do this in map-reduce?

23.6 SQL Group-by using map reduce

We first implement a `group-by`, which groups all rows that have the same key and performs an aggregation function on remaining columns. A map reduce is just that. To implement a `group-by` using map reduce, all we have to do is

Algorithm 18: group by using map reduce

1 emit the group-by column as key
2 perform desired aggregation function on non-group-by columns in the reducer

Let's walk through an example. Suppose we have the following table.

a	b
1	1
1	2
3	3
4	5
5	11

We wish to `group-by` a and sum elements in column b. How can we do this in map reduce? Each row assumed to be an input to a mapper.

```
Group-by a
   sum(b)
```

So, a map reduce is essentially a group-by; they're effectively the same.

23.7 SQL (inner) join using map reduce

This is much more difficult than a `group-by`. There are several cases:

- both tables are so large that neither will fit on a single machine, and
- one table is small enough to fit locally.[50]

The implementations for these two cases are very distinct. In our join, we need to respect the duplicity of the keys. I.e. we must perform the cross-product with all entries appearing in the other table with the same key.

The case when one table fits in memory Suppose we want to merge tables T_1 and T_2, where table T_1 is small enough to fit in memory. For the sake of discussion, let's suppose that each table has two columns.

$$T_1 = \begin{array}{|c|c|}\hline \text{a} & \text{b} \\ \hline \end{array}, \qquad T_2 = \begin{array}{|c|c|}\hline \text{i} & \text{j} \\ \hline \end{array}$$

Do we even need a cluster to perform this? Yes, we definitely need a distributed environment, since the resulting join could be larger than table B. But do we need to perform our sorting operation on our tables? No, we will see that we actually don't need this.

We first broadcast T_1 to all machines via a bit-torrent broadcast. Now, each machine has a full-on copy of table T_1. For fast look-up, we place T_1 into a hash-table.

Algorithm 19: (Hash or Broadcast) Inner Join

1 Place T_1 in a hash-table for fast look-up
2 Place hashed T_1 on each machine via a broadcast
3 Perform an inner join of whatever data in T_2 is stored locally on each machine with its local copy of (hashed) T_1

Each machine does a join of what it owns from T_2 with *all-of* T_1. We don't even need to communicate this result to other machines, we can just leave the merged data local. The inner join that is performed locally on each machine can be done just as we did on the midterm. We assume that the resulting table can be fit on each machine; in the event that we have a cross-product which blows up our table, we would just start assuming that T_1 so large it can't fit on a single machine. We cover this case next, i.e. the case where neither table can fit locally.

23.8 Computations on Matrices with Spark

23.8.1 Distributed Matrices

In Spark, matrices are typically stored broken up for storage in three different ways:

[50] A third case is actually that both tables can fit on a single machine. This is covered by our PRAM model, and we saw this exact question on the midterm.

- By entries (CoordinateMatrix): stored as a list of $(i, j, value)$ tuples
- By rows (RowMatrix): each row is stored separately (e.g. Pagerank)
- By blocks (BlockMatrix): by storing submatrices of a matrix as dense matrices, block matrices can take advantage of low-level linear algebra library for operations like multiplications.

23.8.2 RowMatrix × LocalMatrix

When multiplying a RowMatrix with a small local matrix, we broadcast the entire small matrix to each machine that contains different parts of the RowMatrix and perform multiplications on each machine. Currently, Spark uses BLAS level 1 optimization, which optimizes for vector-vector multiplications during the multiplication.

$$\text{rows are distributed} \left\{ \begin{bmatrix} — & r_1^T & — \\ — & r_2^T & — \\ & \vdots & \\ — & r_n^T & — \end{bmatrix} \right. \begin{bmatrix} | & | & & | \\ l_1 & l_2 & \dots & l_m \\ | & | & & | \end{bmatrix}$$

23.8.3 CoordinateMatrix × CoordinateMatrix

CoordinateMatrix is used to represent sparse matrices, and is stored as a list of (row, column, value) entries. To perform matrix multiplication of two Coordinate Matrix elements $C = AB$, one can summarize the procedure as follows:

Algorithm 20: CoordinateMatrix Multiplication

input: $A : \{(i, j, A_{ij}) | A_{ij} \neq 0\}, B : \{(i, j, B_{ij}) | B_{ij} \neq 0\}$
$J \leftarrow$ Join A, B on $a.j$ and $b.i$ for $a \in A, b \in B$
$M \leftarrow$ For each $j \in J$, map it to (key, value) where key=$(a.i, b.j)$ and value=$a.val \times b.val$
$C \leftarrow$ Reduce M with "+"

Unfortunately, Spark does not have CoordinateMatrix multiplication implemented in the current library. One possible implementation with Scala, when assuming the matrices are stored as `RDD[MatrixEntry(i, j, value)]` is shown below

```
    import org.apache.spark.mllib.linalg.distributed._

val n = 10          // one dimension of the matrix
val range = sc.parallelize({1 to n * n})
// Generate two random sparse matrices of size nxn
val A = range.sample(false, 0.2).map(i => MatrixEntry(i / n, i % n, i))
val B = range.sample(false, 0.2).map(i => MatrixEntry(i / n, i % n, i))

// Perform multiplication
```

```
val C = A.map(e => (e.j, e)).join(B.map(e => (e.i, e)))
         .map(p => ((p._2._1.i, p._2._2.j), p._2._1.value * p._2._2.value))
         .reduceByKey(_ + _).map(p => MatrixEntry(p._1._1, p._1._2, p._2))
```

Effectively, for each $1 \leq i \leq n$, we first join all entries of matrix A on the i-th column with the entries of matrix B on the i-th row, which would create a Cartesian product of two sets of entries for each i. The resulting set contains all possible pairs of entries that would've been multiplied together during a normal matrix multiplication, and each pair of entries is keyed by their shared dimension during the dot product operation. We then remap each element in this set by its position in the result matrix and change its value to the product of the two entries and then reduce each result position with the addition operator. This effectively simulates the dot product operation. Finally, we remap the result to the desired format.

23.8.4 BlockMatrix × BlockMatrix

In some cases, we'd like to multiply two dense matrices for which the rows and columns may themselves be too large to fit in memory on a single machine. By partitioning our matrices into blocks that do fit on a single machine - encoding each one as a BlockMatrix - and performing multiplication on their partitions, we can manage to perform matrix computation on the larger matrices. Using BlockMatrix, we also have the ability to push down the smaller block matrix multiplications to the CPU/GPU directly using Basic Linear Algebra Subprograms (BLAS) routines - as mentioned above, Spark currently uses BLAS level 1 for matrix multiplication. To perform block matrix multiplication, we partition both matrices appropriately so their blocks are equally sized within each matrix, aligned in size across matrices, and so that a single block from each matrix fits together on a single machine. Following partitioning, block multiplication proceeds similarly to coordinate multiplication. Each matrix is flatmapped to produce a list of blocks for multiplication - each block in the first matrix A is effectively copied as many times as the number of columns in the second matrix B, and each block in B is effectively copied as many times as the number of rows in A. Following the flatmap, a cogroup is used to send pairs of complementary blocks that will need to be multiplied to an individual machine, where the multiplication is pushed down to the CPU/GPU level using BLAS. Finally, results of the individual block multiplications corresponding to each block entry in the resulting matrix are sent to the same machine with ReduceByKey and summed up. A simplified version of the Spark code for block matrix multiplication is presented below:

```
def multiply(other: BlockMatrix): BlockMatrix = {

      // Get partitions
      val resultPartitioner = GridPartitioner(numRowBlocks, other.numColBlocks,
        math.max(blocks.partitions.length, other.blocks.partitions.length))

      // Each block of A must be multiplied with the corresponding blocks
      // in each column of B.
```

```
      val flatA = blocks.flatMap {
        case ((blockRowIndex, blockColIndex), block) =>
            Iterator.tabulate(other.numColBlocks)
                (j => ((blockRowIndex, j, blockColIndex), block))
      }

      // Each block of B must be multiplied with the corresponding blocks
      // in each row of A.
      val flatB = other.blocks.flatMap {
        case ((blockRowIndex, blockColIndex), block) =>
            Iterator.tabulate(numRowBlocks)
                (i => ((i, blockColIndex, blockRowIndex), block))
      }

      // Cogroup and multiply block pairs
      val newBlocks: RDD[MatrixBlock] = flatA.cogroup(flatB, resultPartitioner)
        .flatMap { case ((blockRowIndex, blockColIndex, _), (a, b)) =>
          if (a.nonEmpty && b.nonEmpty) {
            val C = b.head match {
              case dense: DenseMatrix => a.head.multiply(dense) // Uses BLAS 1
              case sparse: SparseMatrix => a.head.multiply(sparse.toDense)
            }
            Iterator(((blockRowIndex, blockColIndex), C.toBreeze))
          } else {
            Iterator()
          }
      }
      // Sum up matrices for each block entry of C
      .reduceByKey(resultPartitioner, (a, b) => a + b)
        .mapValues(Matrices.fromBreeze)
}
```

24 Page Rank

We are given as input a directed graph $G = (V, E)$ representing a network of websites. The goal is to find a good metric for measuring the importance of each node in the graph (corresponding to a ranking over the websites). In academia, a paper's importance can be roughly measured by how many citations it receives, and we can use an analogous approach with the web. A website's importance will be measured by the number of sites that link to it. Ideally, we would like the amount of importance conferred on a website by receiving a link to be proportional to the importance of the website giving the link.

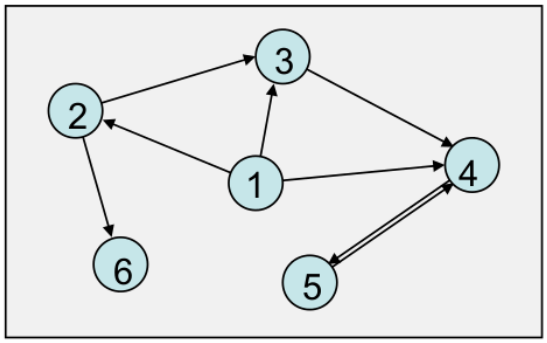


Figure 19: Example network with nodes representing websites and edges representing links

We can formalize this intuition via the process of a random walk. We would like to model a "random-surfer" who is traversing the web with uniform probability of following any link outgoing from the page the surfer is currently on. We are interested in the behavior of this random surfer in the limit as she takes an infinite number of jumps.

To express this in linear algebra, we will make a use of the adjacency matrix, A, and a out-degree matrix D, where:

$$A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

$$D = \begin{pmatrix} deg(v_1) & 0 & \dots & 0 \\ 0 & deg(v_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & deg(v_n) \end{pmatrix}$$

Let $Q = D^{-1}A$. This forms the transition matrix of the random-walker. Given the current state of the walker each row of the matrix gives the probability the walker will transition to each new state.

$$Q_{i,j} = \begin{cases} 1/deg(v_i) & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

It's interesting to note that Q_{ij}^k is equal to the probability of going from node i to node j in exactly k steps, in a random walk over graph G.

The stationary distribution of the Markov Chain with the transition probabilities defined by Q is the solution to our PageRank problem as defined above. The stationary distribution specifies what proportion of time on average is spent at specific node during an infinitely long random walk.

One issue with this approach is the fact that chain can 'get stuck' at the nodes that have no outgoing links, and the nodes with no incoming links are never visited in the random walk. This problem can be fixed by giving our random surfer some "teleportation" probability. Intuitively, the random surfer chooses either to follow a link on the current page or to type a page URL into their browser at random. This amounts to adding a matrix Λ with all positive entries to Q. Naively, Λ could consist of all ones, but by changing the weights in Λ we can encode user preferences (perhaps they're more likely to randomly jump to Reddit than a Machine Learning blog, or vice versa).

This means we now will use the following matrix to parameterize our Markov chain describing our random walker.

$$P = \alpha\Lambda + (1 - \alpha)Q$$

where $0 < \alpha < 1$ is the teleport probability, and Λ is a rank 1 matrix whose rows correspond to the distribution of where a teleporting surfer arrives.

We are looking for a row vector π of node probabilities such that:

$$\pi P = \pi$$

We normally solve problems like this via power iteration. That algorithm is very simple. We initialize $V^{(0)}$ to any initial distribution -for example to the uniform vector $V^{(0)} = [1/n, \ldots, 1/n]$. Then we follow a converging recurrence:

$$V^{(k+1)} = V^{(k)}P = V^{(0)}P^{k+1}$$

If $\lim_{k\to\infty} P^k = P^\infty$ exists, then the previous recurrence converges to the steady-state distribution. However, each recurrence could take a very long time, since V is a very large vector (the number of sites on the internet), and P is a *very* large matrix (the number of sites on the internet *squared*). We will address that more in a moment.

There is a very convenient theorem here that says that this will converge for tractably small k, as a result of the random jump probabilities.

$$\|V^{(k)} - \pi\|_2 \leq e^{-ak}$$

Where $a \approx 2$. This means that with every iteration, we gain roughly another decimal point of precision. At a very high level, this is possible because adding the random jump probabilities (Λ) acts as a strong regularizer on our Markov chain and causes it to mix very quickly. So for $k \geq 8$, we can guarantee very accurate convergence (to around 8 decimal places). This will make PageRank a tractable algorithm, even though every iteration will require us to perform a web-scale matrix-vector multiply.

Note: we do not normalize the vectors at each iteration because we only need the rank ordering in PageRank and normalizing would add complexity. If we were to carry out a large number of iterations the norm of our computed vectors would grow to infinity, however because we restrict ourselves to 8 iterations, this is not a concern.

24.1 Partitioning

The motivation for partitioning arises from the fact that sending data via the network is expensive. The hope is that smartly partitioning an RDD in its constituent machines will reduce the need for communication. We highlight the need for partitioning by highlighting an example through the *PageRank* algorithm.

24.1.1 Sparse Pagerank

The problem is to compute node importance given

- RDD of sparse graph `links` which stores the pair inscribing directed edges.
- RDD storing the rank of the nodes. In this situation, we consider guess for node importance which we refine through iterations.

An application is computing importance of urls (nodes) in the internet(graph). One way is to compute the topmost eigenvalue of the adjacency graph matrix A. Let x_t be the guess of the rank. One way to compute this:

$$x_{t+1} \leftarrow Ax_t.$$

Here, it is assumed that entries of the adjacency matrix are 1 if there is an outgoing link and this is normalized by the number of neighbors the node has. We use the following modification:

$$x_{t+1} \leftarrow 0.15\mathbf{1} + 0.85Ax_t.$$

In other words,

1. Each page (node) is started with rank 1. Or x_t is initialized with the uniform vector.
2. On each iteration page p computes its contribution $(x_t)_p/|\text{neighbors}_p|$ and sends it to its neighbors.
3. Each page's rank is $0.15 + 0.85 \times$ contribs. We are implementing this matrix-vector multiplication using `join` and aligning the RDDs.

The algorithm is now written:

```
val links =  // RDD of (url, neighbors) pairs
var ranks =  RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
        val contribs = links. join(ranks). flatMap {
```

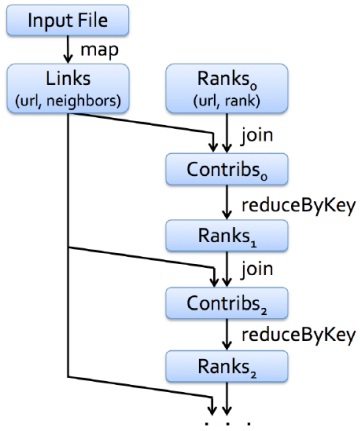


Figure 20: Pagerank without partitioning incurs multiple all-to-all communication from repeated joins.

```
                case (url,(links,rank)) => links.map(dest => (dest,rank/links.size))
        }
        ranks = contribs. reduceByKey(_+_). mapValues(.15+0.85*_)
}
```

The parts in red are local operations and the parts in blue are distributed operations. *All-to-all* communication is required in the `join` and `reduceByKey` operations. As can be seen from Fig. 20, joins necessitating all-to-all communication are required for each iteration.

We *partition* the links such that all links within the same partition sit on the same machine. This can be done with the following line-

```
val links = sc.textfile(...)....partitionBy(new HashPartitioner(8))
```

Any shuffle operation (such as one with join) will respect the partitioner if set. As can be seen from Fig. 21, new ranks after the first iteration sit in the same machine as the old links and repeated joins do not occur.

A note of caution here is that `map` may change key values and partitioning is not respected. If knowledge of partitioning is to be preserved, `mapValue` can be employed.

Further communication can be cut down from domain knowledge. For example, we know that most links from a webpage are to another page with the same domain name. Thus we can partition all webpages with the same domain in the same machine.

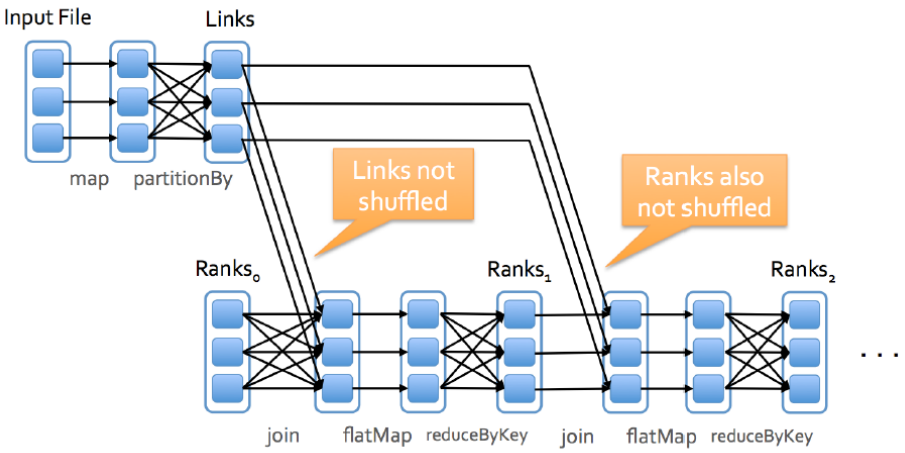


Figure 21: With partitioning, repeated joins does not happen.

25 Optimization, scaling, and gradient descent in Spark

We will now consider two different optimization algorithms: gradient descent, and stochastic gradient descent. In this class, we'll implement gradient descent in Spark, and then see why SGD not suitable for Spark. Then we'll learn how to implementSGD regardless of the data-flow paradigm of the language we're computing in. For the remainder of this section, we will assume we have a separable objective functions, of the form

$$\min_x \sum_{i=1}^{n} F_i(x)$$

where $w \in \mathbb{R}^d$ and F_i is the loss-function applied to a training point i. We won't make any assumption about the specific form of each F_i.

The main question we are interested in is how our algorithm scales both in n, the number of data points we have, and d the dimension of our model. In the first part of the lecture, we'll assume n is too large to fit on a single machine but that d numbers can fit on a single machine's RAM. Recall that gradient descent starts with a random initial vector, x_0, say initialized to all zeros. Then at each iteration $x_{k+1} \leftarrow x_k - \alpha \sum_{i=1}^{n} \nabla F_i(x_k)$, where $\nabla F_i(\cdot)$ is a vector itself.

25.0.1 Implementing Gradient Descent in Spark

The following block of code demonstrates an implementation of gradient descent in Spark:

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
For (i <- 1 to numIterations):
  val gradient = points.map(p => $\nabla F_p(w)$) .reduce(_+_)
  w -= alpha * gradient
```

The first line reads in a text-file of training points and their associated labels. The operation, `parsePoint` is a closure which takes as argument a single like of the text-file of training points and their associated labels, and outputs a clean d-dimensional x_i vector and a single label y_i. After parsing the files, we cache the RDD with the training points. The action `cache` tells Spark that we're going to use a particular RDD frequently, so each machine should try to hold that its part of the RDD in memory as much as possible rather than flushing data to disk. Since we need to access this RDD every iteration of gradient descent, we will want to keep it in memory. Recall that `cache` is an action (as opposed to a transformation), so Spark actually kicks off computation and reads in the text file, applying the `parsePoint` closure to each part of the distributed file.[51] The next line initializes the model parameters to a d-dimensional vector of all zeros. Finally, each iteration

[51] Suppose we have a 1 terabyte dataset, and we have a small amount of memory, say 10 gigabytes. Suppose we wish to process the large dataset. We can read directly off the large terabyte slowly, or we can read directly off the ram for faster access. The question becomes which part of the data set to store in ram since we can't fit the entire object in memory. We commonly employ Least Recently Used caching, which evicts items from ram according to which piece of data was least recently used. This is a simple heuristic, but works in practice.

of the for loop implements an iteration of gradient descent. The closure

$$\texttt{points.map(p => } \nabla F_p(w)\texttt{)}$$

maps each of the training points to their associated gradients of the loss function. Specifically, $\nabla F_p(w)$ is the gradient of F_p evaluated at point w. For example, if $F_i(w) = (w^T x_i - y_i)^2$, then ∇F_i may be calculated symbolically, and we may plug this formula into $\nabla F_p(\cdot)$. Then we calculate the full gradient as the sum of the individual gradients—we use reduce for this. Recall that Spark uses lazy evaluations, the the computation is not kicked off until this reduce step. Under the hood, Spark takes into account communication cost between machines, scheduling, and executing operations locally in a PRAM model.

25.0.2 Broadcasting in Spark

Recall that by default, when a `map` happens, anything that is needed for the `map` to happen is shipped out to the workers. So in the line

$$\texttt{points.map(p => } \nabla F_p(w)\texttt{).reduce(\_+\_)}$$

since the function that maps each training point to it gradient depends on w, a copy of w will be sent to **every** CPU of **every** machine. This happens, because by default, we assume w is being modified. Consequently, we must store w separately to avoid any concurrent write issues. However, when we calculate each gradient step, w is not being modified so there is no chance of a concurrent write. We are wasting storage by not storing w in shared memory. To get around this, we can broadcast `w` so that it only gets sent to each machine a single time. To do this, we define `w` as a broadcast variable. The following block of code illustrates gradient descent with the model parameters as a broadcast variable.

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
var w_br = sc.broadcast(w)
For (i <- 1 to numIterations):
  val gradient = points.map(p => $\nabla F_p$(w_br.value)) .reduce(_+_)
  w -= alpha * gradient
  w_br = sc.broadcast(w)
End
```

When we initialize w, we first declare it as a broadcast variable. In each of the iterations of gradient descent, we need to refer to the *value* of w. Finally, we re-broadcast w after it gets updated. In addition to being more storage efficient on each machine, i.e. storing a single copy of the data object instead of multiple, we also take advantage of bit-torrent broadcasting when we use Spark's broadcasting.

25.1 Analysis

Recall, gradient descent requires $\mathcal{O}(\log \frac{1}{\epsilon})$ iterations to achieve an ϵ-optimal solution if f is L-smooth, and μ-strongly convex. While the computation of the gradients can be distributed across the cluster of machines, the iterations of gradient descent are inherently serial. At each iteration, Spark implements a synchronization barrier to ensure that w is fully updated before the workers begin computing the next iteration. Just as in the parallel case, if there are stragglers, the other workers are idle waiting for the next iteration. The depth of the computation of typically is the bottleneck rather than the gradient updates.

Communication time The map is embarrassingly parallel, and requires no communication. The broadcast requires one to all communication, and the reduce requires all-to-one communication. So, per iteration, our total communication time scales with

$$2\log_2(p)(L + \frac{m}{B})$$

[52]

25.1.1 SGD

What about SGD? In stochastic gradient descent, we sample a training point from $i = 1, \ldots, n$ to get a point-estimate of the gradient $\nabla F_i(w)$, and then we update $x_{k+1} - = \alpha \nabla F_i(x_k)$. So instead of using the full summation, we only look at a single point. However, with SGD we need more iterations, which increases the depth. SGD was still a viable solution in the parallel setting because we could use HogWild! and not wait between iterations for updates to be written to w. However, implementing HOGWILD! in Spark is difficult because Spark has synchronization barriers: we have to wait for the broadcast to finish, then the map and reduce to finish, then another broadcast. There is no way to override these synchronization barriers because the RDDs must be fault-tolerant, and it is very difficult to track the lineage of an RDD if you cannot control when it is being updated. Consequently, SGD is not the right tool for SGD and Hogwild!

What is the right tool for SGD? Tools such as Tensorflow allow us to set up clusters on our own that do not have fault tolerance or synchronization barriers. To implement SGD, we set up a parameter server that holds w, and each worker asks the parameter server for a current copy of w. Then, the machine samples its own data, computes a gradient, and sends the resulting update back to the driver. Using this setup, it is typically does not derail our algorithm if a machine goes down. We won't get updates from that part of the training data for some time, but we can detect that the machine goes down and restart computation. With the parameter server, we need to checkpoint

[52] Technically, when we call `points.map()`, we serialize code and send it to each machine. One reason we chose Scala to implement Spark is that it's easy to serialize code. We remark that there is a function called `allReduce` which merges the concept of a reduce and broadcast. After a reduce, the result of the reduce ends up on the driver machine. After an `allReduce` the result of the `reduce` gets sent to all machines via a broadcast; the broadcast happens simultaneously with the regular reduce.

our results once in a while to ensure that if it dies, we don't have to re-do too much work. However, optimization lends itself well to warm-starts, meaning that we do not need to exactly recreate our iterates (making RDDS and Spark inefficient).

26 Complexity measures for MapReduce

During a MapReduce algorithm, three main stages are involved while computing the complexity. First, all of the map tasks have to finish. Second, all of the pairs which need to be on the same machine need to be shuffled. Finally, reducers take some time to finish. As a consequence, we will provide three complexity time measures corresponding to these three stages:

- **Mappers cost time:** the time the map phase takes. All the mappers have to do their computations and spit out the pairs (this is embarrassingly parallel work, so we can use the same analysis tools that we know for a single processor computation and take the worst one).
- **Shuffle cost time:** number of items the mappers output to be sorted: how many tuples have to be sorted. Note that although the keys are hashable, you can not just assume that the sort time will be $O(n)$ as in practice, a more complex analysis involving the number of machines and the architecture would have to be done.
- **Reducers cost time:** how long it takes for the slowest reducer to finish.

Any of these measures could be the bottleneck of the overall analysis of a MapReduce algorithm.

27 Triangle Counting

For this part, assume that we are given an undirected graph $G(V, E)$ in the edge-list format, i.e. a giant data set of (u, v) pairs, split across machines. Let's make the following assumption: the number of nodes of G fits in memory. For example, $n \simeq 10 \cdot 10^6$. However, the number of edges $m \simeq n^2$ (for a dense graph) does not fit in memory. As a consequence, the node data structure can be thought as a "local array", whereas the edge data structure can be thought of as an "RDD".

Goal: Counting the number of triangles in the graph within a MapReduce implementation.

27.1 Triangle counting on a single machine

First, let us derive a sequential algorithm to count every triangle of our graph G in a sequential way. Let us use the adjacency matrix A of the graph: $A = (a_{ij})$ where

$$a_{ij} = \begin{cases} 1 \text{ if (i,j)} \in \text{E} \\ \\ 0 \text{ otherwise} \end{cases}$$

Then, it is well known that the global coefficient $[A^k]_{ij}$ for $k \in \mathbb{N}$ counts the number of paths of length k in G, starting at i and ending at j. In particular $\forall i \in V$, $[A^3]_{ii}$ counts the number of paths starting at i and ending at i of length 3, i.e. the number of triangles containing i as a vertex. To avoid overcounting, we need to divide $\sum_i [A^3]_{ii}$ by a factor of 6: indeed a single triangle will be counted $2 * 3 = 6$ times. The factor 2 comes from the fact that the triangle (u, v, w) can be traveled as $u \to v \to w \to u$ or $u \to w \to v \to u$.

What is the time complexity? We know that we can compute A^3 in $O(n^\omega)$ with $\omega \simeq 2.373$, but this algorithm cannot be distributed. Likewise with Strassen's algorithm where $\omega \simeq 2.8$, it can not be adapted to a cluster because it would need to send giant submatrices of data across machines. As a general idea, it can be noted that recursive algorithms are not as useful for clusters as they are for the PRAM model.

27.2 Triangle counting on a cluster: the Node Iterator Algorithm

So let us do something simpler for the distributed case. Note that if the graph is sparse, even on a sequential machine you would want to do better than $O(n^{2.373})$. The idea of the Node Iterator Algorithm is the following: given our graph $G(V, E)$, one needs to iterate through each vertex $v \in V$ to find its neighborhood $\Gamma(v)$. Then, for all $u, w \in \Gamma(v)$, check if (u,w) is in E. If it is the case, then we have found a legitimate triangle, and an accumulator variable T can be incremented.

Algorithm 21: *NodeIterator*$(G(V, E))$

```
procedure NodeIterator( )
    T ← 0
    for v ∈ V do
        for u ∈ Γ(v) do
            for w ∈ Γ(v) do
                if (u, w) ∈ E then
                    T := T + 1/6
                end
            end
        end
    end
    return T
end procedure
```

Let us consider the time complexity of this algorithm: On a sequential machine, it is at most $O(n^3)$ (if the graph is dense for example, there are at most $O(n^3)$ triangles). But here, we would like to exploit sparsity, i.e. to rewrite this complexity as a function of m. Now let's say that the graph is sparse with m edges. Let's say there is a high degree node v, which means that v is roughly connected to all the other nodes. (v is very popular). Suppose:

$$d(v) > cn \text{ where } c < 1$$

Then, the runtime is going to be $\Omega(n^2)$. Actually, it will take $\Omega(n^2)$ time just for that single node v.

Let's now implement this algorithm in a MapReduce environment.

28 Implementing the Node Iterator Algorithm in MapReduce

The Node Iterator algorithm can be implemented in two MapReduce steps.

28.1 Compute neighborhoods

Recall that we are given an edge list, but that the algorithm requires a list of neighbors for each node in order to iterate over.

Neighborhood collection can be performed in a straightforward MapReduce step, with mappers emitting the first element of the edge (u, v) as the key and the reducers collecting the neighborhoods $\{v_1, v_2, ..., v_n\}$ corresponding to each key u. This leaves us with an **RDD**[(Int, Array[Int])] object.

Note that each list of neighbors is of size at most n, meaning that by assumption it fits in memory as is required for individual elements of RDDs. Note as well that in Spark, we could implement this operation using *groupByKey*.

28.2 Count triangles

The key to accomplishing the counting step in a single MapReduce step is observing that we need the edges to search for *and* the actual edges in the graph both available to the reducer to perform the lookup for whether given triangles exist.

We can do this by having mappers (a) loop over all neighbors u and w for each node $v \in V$ and output key-value pairs [(u, w), "to check"] and (b) traverse the edge list and output the pairs [(v_1, v_2), "present edge"]. That is to say the keys are edges and the values are whether the particular edge is being searched for or represents the actual edges in the graph.

The reducer then has access to, for each edge, a list consisting of zero or more "to check" strings and zero or one "present edge" strings. If there exists a "present edge" string in the list, then the reducer sums the number of "to check" strings it finds, as each represents a discovered triange. Otherwise, that edge either doesn't exist in the graph or wouldn't complete a triangle. Note that the sum, keeping in line with the algorithm, would be divided by six to avoid counting triangles multiple times.

29 Complexity Analysis of MapReduce Algorithm

We focus on the second step of the algorithm and leave an analysis of the first step to the reader.

Recall that analyzing the runtime of a MapReduce operation requires going over each phase: map, shuffle, and reduce.

1. **Map Phase**: The time to process a single neighborhood is $O(n^2)$, with the upper bound coming from a node with $O(n)$ neighbors.

2. **Shuffle Size**: The map phase emits $\Omega(n^2 + m)$ objects in the case of a single dense node with $O(n)$ neighbors. In a dense graph, each node has $O(n)$ neighbors and the shuffle size is $O(n^3 + m)$.

Note that we only consider the shuffle size here, not the cost of the actual shuffle. Partly, this is because we can achieve a linear time sort due to the hashable objects being sorted, albeit with a high constant factor. In addition, the sorting algorithm's asymptotic performance can vary by implementation, while shuffle size is independent of those concerns.

3. **Reduce Phase**: Processing a list of "to check" and "present edge" strings is $O(n)$ in the case that each endpoint of the edge (v_1, v_2) is connected to O(n) other nodes.

30 Improving the Node Iterator Algorithm

Even if the above algorithm were $O(n^2)$ rather than $O(n^3)$, it still would not be good enough to work with large graphs. For example, if $n = 10$ million, $n^2 = 100$ trillion. Ideally, we'd like a dependence on m instead, especially when the graph is sparse.

To get there, we combine two insights. The first is that the driver of high costs is nodes with large neighborhoods, which then have to output all pairs of nodes in the neighborhood. The second is that our current algorithm is substantially overcounting triangles by starting its search from each endpoint of the triangle. For example, given a triangle with endpoints a, b, and c, the algorithm will detect that b and c are in the neighborhood of a, a and c are in the neighborhood of b, and a and b are in the neighborhood of c.

We address each of these shortcomings by pursuing a strategy to only search for a given triangle T in the neighborhood of its least-degree node. To do this, create a total ordering on nodes by degree (i.e. break ties in some way) and remove all nodes from the neighborhood of a node v with a lower degree than v.

By construction, this leaves us with a set of neighborhoods $\Gamma^*(v)$ where each node v is only neighbors with higher-degree nodes. It is now apparent that when searching for triangles by looping over these neighborhoods instead, triangles will only be found from the neighborhood of the least-degree node as we wanted.

It turns out that this approach yields a runtime of $O(m^{3/2})$.

Formally, let $\succ$ be a total order on all vertices such that $u \succ v$ if and only if $\deg(u) > \deg(v)$, with ties broken arbitrarily but consistently. How ties are broken is not particularly important as long as the ordering remains consistent between runs. We can use this ordering to reduce the amount of over-counting and therefore work done by NodeIterator.

Define the modified neighborhood of v, $\Gamma^*(v)$, as $\{u \in \Gamma(v) \mid u \succ v\}$. Roughly speaking, this is the set of neighbors of v with higher degree than v, and we can use these modified neighborhoods when counting triangles so that only the lowest degree node in each triangle 'counts' the triangle. Then, instead of counting each triangle 6 times we only count each triangle twice.

If we fix a arbitrary number $t > 0$, the complexity of this algorithm is upper bounded by

$$\sum_{v \in V} \binom{\deg(v)}{2} = \sum_{v \in V, \deg(v) \geq t} \binom{\deg(v)}{2} + \sum_{v \in V, \deg(v) < t} \binom{\deg(v)}{2}.$$

The expansion separates the summation for high degree nodes ($\deg(v) \geq t$) and low degree nodes. By the handshake lemma, we know that the number of high degree nodes is at most $\frac{2m}{t}$ (there are

Algorithm 22: NodeIterator++(V,E)

```
T ← 0
for v ∈ V do
    for u ∈ Γ*(v) do
        for w ∈ Γ*(v) do
            if (u, w) ∈ E then
                T ← T + 1/2
            end
        end
    end
end
return T
```

m edges each contributing to 2 degrees). Therefore, the first part of the sum is at most $(\frac{2m}{t})^3$ (# high degree nodes choose 3)

For the low degree nodes (i.e. nodes v such that $\deg(v) < t$) we have that

$$\sum_{v \in V, \deg(v) < t} \binom{\deg(v)}{2} \leq \sum_{v \in V, \deg(v) < t} \deg(v)^2 \leq t \left(\sum_{v \in V, \deg(v) \leq t} \deg(v) \right) \leq 2mt.$$

This implies that the total work done by NodeIterator++ is upper bounded by $(\frac{2m}{t})^3 + 2mt$. We can choose $t = \sqrt{m}$ to minimize this, giving $4m^{3/2} + 2m^{3/2} = O(m^{3/2})$ runtime for NodeIterator++. For sparse matrices, $O(m^{\frac{3}{2}}) < O(n^{2.373\ldots})$, so we have achieved our goal of improving the matrix multiplication algorithm by exploiting graph sparsity.

A reasonable question to ask is whether or not we lost some rigor or sharpness by changing the algorithm in this way. We can construct an example such that there is a clique $K_{\sqrt{n}}$ and $n - \sqrt{n}$ nodes that are connected in a line (see figure 30). In this example, the number of triangles is $\Theta(\binom{\sqrt{n}}{3}) = \Theta(n^{\frac{3}{2}})$ and $m = \Theta(\binom{\sqrt{n}}{2}) = O(n)$. Thus, our algorithm matches up with this bound. One important concept that this example illustrates is that there is a noticeable difference in complexity between listing and counting triangles. In order to list the triangles in a clique, there is $O(n^3)$ work as there are just that many triangles. However, if we were just counting, we could do it faster by just calculating $A^2 + A$, which is just $O(n^{2.37})$ amount of work.

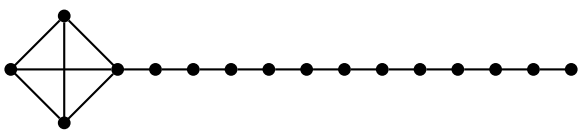


Figure 22: An example of a lollipop graph

31 Singular Value Decomposition (SVD)

Today we're going to see how to do SVD in a distributed environment where the matrix is split up across machines row by row[53]. Recall that the rank-r singular value decomposition (SVD) is a factorization of a real matrix $A \in \mathbf{R}^{m \times n}$, such that $A = U\Sigma V^T$, where $U \in \mathbf{R}^{m \times r}$ and $V \in \mathbf{R}^{n \times r}$ are unitary matrices holding the left and right singular vectors of A, and Σ is a $r \times r$ diagonal matrix with non-negative real entries, holding the top r singular values of A. Recall that U, V being unitary means that $U^T U = V^T V = I$, i.e. its transpose is its inverse.

Large data, few features For this lecture, we will focus on row matrices that are tall and skinny. Specifically, if A is $m \times n$ then $m \gg n$. We will assume that n^2 fits in memory on a single machine. For example, our data could be one trillion movies and each has a thousand features such as text-transcription and director, acting staff, etc. Computing full-on SVD requires $O(mn^2)$ work. Computing the top k singular values and vectors costs $O(mk^2)$ work. This is still a tremendous amount of work even on a cluster.

31.1 When A is a `RowMatrix`

We will explicitly use the assumption that our matrix is tall and skinny. The following computation shows that the singular values and right singular vectors can be recovered from the SVD of the Gramian matrix $A^T A$:

$$A^T A = (U\Sigma V^T)^T U\Sigma V^T = V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T$$

We can exploit this property to efficiently compute the SVD of a tall-skinny matrix. If n is small enough to fit on a single machine, then A can be distributed as a one-dimensional block-row matrix (in Spark this is called a `RowMatrix`). (We will hold off on details of computing $A^T A$ until the following section.) We may then solve the SVD of $A^T A$ locally on a single machine to determine Σ and V. Finally, we can solve for U by simple matrix multiplications:

$$A = U\Sigma V^T \implies U = AV\Sigma^{-1}$$

So our general approach is as follows

We'll explain the implementation of each step in the MapReduce framework in detail. Note that A is stored as a row matrix, and entries of $A^T A$ are all pairs of inner-products between columns of A.

[53] We have many different ways of representing a matrix: coordinates are sprinkled across machines, or rows can be sprinkled across machines, and then there's the block matrix which is itself a lot like a coordinate representation except each coordinate represents a matrix rather than an element.

Algorithm 23: Distributed SVD

1 Compute $A^TA = V\Sigma^2V^T$ `// dimension` $n \times n$
2 Compute top k singular values of A^TA on a single machine `// using local LinAlg ops`
3 Compute $U = AV\Sigma^{-1}$ `// distributed multiplication`

31.2 Computing A^TA

A is an $m \times n$ matrix:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

Note that A has sparse rows, each of which has at most L nonzeros. In general, A is stored across hundreds of machines and cannot be streamed through a single machine. In particular, even two consecutive rows may not be on the same machine. An example of A in real applications would be a Netflix matrix: A lot of users and only a few movies. Rows are all sparse, but some column can be very dense, e.g., the column for a popular movie, such as the *Godfather*. Our task is to compute A^TA which is $n \times n$, considerably smaller than A. A^TA is in general dense; each entry is simply a dot product of a pair of columns of A.

Implementing as map-reduce The key observation is that A^TA is a much smaller matrix than A when $n \ll m$. In general, computing the rank-r SVD of A will cost $O(mnr)$ operations (not to mention expensive communication costs), while this computation only costs $O(n^2r)$ operations for A^TA. Similarly, if A is short and fat (i.e. $n \gg m$), we can run this same algorithm on A^T. Since $(A^TA)_{jk} = \sum_{i=1}^{m} a_{ij}a_{ik}$, this gives insight for our mapper.

Algorithm 24: A^TA mapper

Input: The ith row of a matrix, denoted by r_i
1 **for** *all (non-zero) pairs* a_{ij}, a_{ik} *in* r_i **do**
2 Emit $\langle (j,k) \to a_{ij}a_{ik} \rangle$ `// (j,k) is the key, the product is the value`
3 **end**

The reducer is a simple summation.

Computing $U = AV\Sigma^{-1}$ We compute $V\Sigma^{-1}$ on a single machine, since it's only of dimension $n \times k$. This matrix-product can then be broadcast to all machines. After this broadcast, we don't require machines to talk with each other anymore. Let $w = V\Sigma^{-1}$, and compute Aw. We allow

Algorithm 25: A^TA reducer

Input: A coordinate pair as key and a listing of products of scalars: $\big(\langle j,k\rangle, (v_1,\ldots,v_m)\big)$

1 $(j,k) \to \sum_{i=1}^{m} v_i$

the rows of A to stay where they are, and locally compute Aw, and the result sits on each machine as desired in the end.

Communication costs We assume that we use combiners so that each machine locally computes its portion of A^TA in line 1 separately before communication between machines occurs in line 2. The only communication costs are the all-to-one communication on line 2 with message size n^2, and the one-to-all communication on line 5 with message size nr. These require $O(\log p)$ messages when a recursive doubling communication pattern is used. If each row has at most L non-zero entries, then it is easy to see that the shuffle size is $O(mL^2)$ since there are m mappers and each performs $O(L^2)$ emits, and the largest reduce-key is $O(m)$. Since m is usually very large (e.g., 10^{12}), this algorithm would not work well. It turns out that we can bring down both complexities via clever sampling. This leads us to Dimension Independent Matrix Square using MapReduce (DIMSUM), which we will cover in the next section.

Network communication patterns The first mapreduce (to compute A^TA) is a potential all-to-all for the emit stage and all-to-one in the reduce. The second mapreduce (to compute $U = AV\Sigma^{-1}$) is one-to-all.

32 Problem Sets

References

[1] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, *Communication-optimal parallel algorithm for strassen's matrix multiplication*, CoRR, abs/1202.3173 (2012).

[2] M. Blum, *Probabilistic analysis and randomized quicksort*, Carnegie Mellon, 15451-s07, (2007).

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, *Time bounds for selection*, JCSS, (1973), pp. 448–461.

[4] B. Chazelle, *A minimum spanning tree algorithm with inverse-ackermann type complexity*, NECI Research Tech Report, (1999).

[5] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[6] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Computation, 9 (1990), pp. 251–280.

[7] J. Nesetril, *On minimum spanning tree problem (translation)*, Elsevier, (2001).

[8] B. Recht, C. Re, S. Wright, and F. Niu, *Hogwild: A lock-free approach to parallelizing stochastic gradient descent*, NIPS, (2011).

[9] V. Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik, 13 (1969), pp. 354–356.

[10] V. V. Williams, *Multiplying matrices in* $o(n^{2.373})$ *time*, Stanford University, (2014).