Foundations and Trends® in Optimization

Massively Parallel Computation: Algorithms and Applications

Suggested Citation: Sungjin Im, Ravi Kumar, Silvio Lattanzi, Benjamin Moseley and Sergei Vassilvitskii (2023), "Massively Parallel Computation: Algorithms and Applications", Foundations and Trends® in Optimization: Vol. 5, No. 4, pp 340–417. DOI: 10.1561/2400000025.

Sungjin Im
University of California, Merced

Ravi Kumar
Google, Mountain View

Silvio Lattanzi
Google, Barcelona

Benjamin Moseley
Carnegie Mellon University

Sergei Vassilvitskii
Google, New York

now
the essence of knowledge
Boston — Delft

Contents

Massively Parallel Computation: Algorithms and Applications

Sungjin Im[1], Ravi Kumar[2], Silvio Lattanzi[3], Benjamin Moseley[4] and Sergei Vassilvitskii[5]

[1] *University of California, Merced, USA; sim3@ucmerced.edu*
[2] *Google, Mountain View, USA; ravi.k53@gmail.com*
[3] *Google, Barcelona, Spain; silviol@google.com*
[4] *Carnegie Mellon University, USA; moseleyb@andrew.cmu.edu*
[5] *Google, New York, USA; sergeiv@google.com*

ABSTRACT

The algorithms community has been modeling the underlying key features and constraints of massively parallel frameworks and using these models to discover new algorithmic techniques tailored to them. This monograph focuses on the Massively Parallel Model of Computation (MPC) framework, also known as the MapReduce model in the literature. It describes algorithmic tools that have been developed to leverage the unique features of the MPC framework. These tools were chosen for their broad applicability, as they can serve as building blocks to design new algorithms. The monograph is not exhaustive and includes topics such as partitioning and coresets, sample and prune, dynamic programming, round compression, and lower bounds.

Sungjin Im, Ravi Kumar, Silvio Lattanzi, Benjamin Moseley and Sergei Vassilvitskii (2023), “Massively Parallel Computation: Algorithms and Applications”, Foundations and Trends® in Optimization: Vol. 5, No. 4, pp 340–417. DOI: 10.1561/2400000025.

1

Introduction

The modern era is witnessing a revolution in the ability to scale computations to massively large data sets. A key breakthrough in scalability was the introduction of fast and easy-to-use distributed programming models such as MapReduce (Dean and Ghemawat, 2008), Hadoop (hadoop.apache.org), and Spark (spark.apache.org). We refer to these programming models as **massively parallel frameworks**.

Massively parallel frameworks were originally designed for relatively simple types of computations such as counting the frequency of words in a data set. Since then, they have been shown to be useful for a far richer class of applications. The goal of a recent line of work is to study these frameworks algorithmically to unlock their true underlying power and expand their applicability. The hope is, through an algorithmic investigation, to achieve successes similar to those on topics such as cache-oblivious algorithms (Frigo *et al.*, 2012) and data streaming algorithms (McGregor, 2014).

Practically, massively distributed frameworks enable programmers to easily deploy algorithms on tens to thousands of machines. Algorithmically, the frameworks have restrictions on their computational expressive power to help ensure programs can be efficiently parallelized.

The challenges are then to (i) develop simple tools that reveal fundamentals of massive computation and aid algorithm design and (ii) understand which computations can benefit from the framework.

The algorithms community has been addressing this problem by modeling the underlying key features and constraints of massively parallel frameworks and using these models to discover new algorithmic techniques tailored to them. The first model of massively parallel computation was introduced for the MapReduce framework by Karloff *et al.* (2010) and several variants have been proposed since (Feldman *et al.*, 2010; Koutris *et al.*, 2018; Beame *et al.*, 2017; Andoni *et al.*, 2014; Goel and Munagala, 2012; Goodrich *et al.*, 2011; Pietracaprina *et al.*, 2012; Roughgarden *et al.*, 2016). Perhaps the main advantage of the model in Karloff *et al.* (2010) is its relative simplicity. It captures framework characteristics that are sufficient for algorithm design, without delving into the plethora of system parameters. In this monograph, we will primarily focus on this version of the model; we call it the Massively Parallel Model of Computation (MPC). See Section 2 for formal details.

The MPC model is a special case of the Bulk-Synchronous-Parallel (BSP) model of Valiant (1990), where machines have sublinear memory (i.e., n^δ for $\delta < 1$ and input size n) and computation proceeds in alternating **rounds** of communication and sequential computation. The MPC model can be thought of making different trade-offs than the classic PRAM computational model. Much of the difference comes from being able to run a sequential algorithm on a small sublinear portion of the data during a single round. Full details are given in Section 2.

The MPC model has a strong connection to practice and this is demonstrated by algorithmic developments resulting in good practical performance (Chierichetti *et al.*, 2010; Bahmani *et al.*, 2012a; Suri and Vassilvitskii, 2011; Karloff *et al.*, 2010; Mirzasoleiman *et al.*, 2013; Broder *et al.*, 2014; Feldman *et al.*, 2010; Zhao *et al.*, 2012; Ene *et al.*, 2011; Malkomes *et al.*, 2015; Kumar *et al.*, 2015; Bahmani *et al.*, 2012b; Ene and Nguyen, 2015; Cohen-Addad *et al.*, 2021b; Cohen-Addad *et al.*, 2021a; Lattanzi *et al.*, 2019; Ghaffari *et al.*, 2019b; Bateni *et al.*, 2017; Assadi *et al.*, 2019b; Bhaskara and Wijewardena, 2018) and influencing software libraries. For example, theoretical algorithms for k-means clustering have been incorporated in the Spark Machine Learning software library[1] (Bahmani *et al.*, 2012b).

1.1 Purpose of This Monograph

This line of work has demonstrated that massively parallel frameworks are useful for some challenging applications. With this as a proof-of-concept, an exciting area of research is to broaden the use of the frameworks to address a wide range of problems by using theoretical models to drive algorithm design.

This monograph will describe algorithmic tools that have been developed for massively distributed computing that leverage the unique features of the framework. The tools were chosen because we believe they are generally applicable and can be used as building blocks to design algorithms in the area.

This monograph is not exhaustive. However, it will cover the following areas.

- **Partitioning and Coresets:** This is one of the most natural approaches for parallel algorithms design. The idea is to partition the input to the problem across machines, and have each machine solve the problem on the individual parts. The individual solutions are then combined to build the solution to the overall problem.

- **Sample and Prune:** Another common approach to solve problems on large data sets is to use sampling to reduce problem size. Unfortunately, sampling from simple distributions, such as uniform, often misses too much information to solve a problem near optimally. We discuss the iterative sample-and-prune method, which has been shown to be efficient for many problems.

- **Dynamic Programming:** Dynamic programming is a powerful technique for solving problems. Unfortunately, it is typically difficult to parallelize. We discuss techniques for adapting certain dynamic programs to the massively parallel setting.

[1] https://spark.apache.org/docs/2.2.0/mllib-clustering.html

- **Rounds Reduction:** A simulation approach to solve problems in a parallel fashion is to apply a known algorithm, performing one step in a single round of distributed computation. While simple, it is often inefficient and leads to a large number of rounds. We discuss round compression, where multiple iterative rounds are compressed into a single round.
- **Lower Bounds:** Finally, we discuss the limitations of the massively parallel model of computation. We highlight the efforts to develop lower bounds for the model and derive connections to other models of computation.

1.2 Prerequisites

This monograph will assume the basics on approximation algorithm design and randomized algorithms. For a quick overview, we recommend the books by Williamson and Shmoys (2011, Chapter 2) and Mitzenmacher and Upfal (2005, Chapters 1-4).

2

The MPC Model

The goal of this section is to formally introduce the massively parallel computing (MPC) model and compare it to other concrete computational models. In some cases, the MPC model can provably simulate algorithms in other models and vice versa. Further, many of the algorithmic techniques developed for the MPC model have found uses in parallel computing, distributed computing, and data stream algorithms.

Before we introduce the MPC model, we take a step back to consider three characteristics that summarize the massively parallel framework. The first is the frequency and synchronicity of communication: how often do the machines coordinate with each other. The second is the topology of the communication: what are the allowable communication channels between machines? The third is the degree of parallelism: how does the number of parallel tasks grow with the problem size? As we will see below, different answers to these questions lead to different models of computation. For instance, the PRAM model restricts neither the frequency nor the topology, and allows for a super-linear degree of parallelism; on the other hand, the LOCAL model restricts the topology and parallelism.

The different characteristics also have an effect on the difficulty of programming under these conditions. For example, multithreaded programming can be a notoriously tricky endeavor, since different executions of the same piece of code may result in different code paths, leading to code that is very hard to test, and bugs that can be hard to reproduce. Additionally, an often ignored issue is that of system failures: with computation happening concurrently on multiple pieces of hardware, the chance of one of them failing increases dramatically, thus systems must be robust to failures, which leads to even more code complexity.

Indeed, the development of MapReduce was largely motivated by trying to hide as much of the system complexity as possible from the algorithm designer, at the cost of slightly limiting the model of computation. As Dean and Ghemawat (2008) write in their seminal work:

> The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues. As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library.

While the first uses of MapReduce were for simple computations, the framework has proven incredibly versatile and powerful. Researchers have developed new algorithmic techniques tailored to this computational model and MapReduce-style algorithms are now known and implemented for a large number of problems, from clustering, and submodular optimization, to basic graph problems.

2.1 Formal Definition

The MPC paradigm forces the algorithm designer to break up the computation into a sequence of synchronous rounds, where individual tasks must work in isolation during a round, and can communicate with each other only between rounds. Rounds are relatively expensive, and thus the goal is to minimize the overall number of rounds an algorithm takes.

Formally, consider an input of size N. To process it, we have p *machines*, each with an M-bit *memory*. Initially, the input is distributed among the memory of the p machines in an arbitrary manner. For instance, machine 1 has bits 1 through M, machine 2 has bits $M + 1$ through $2M$, and so on. The computation proceeds in rounds. In each round each machine reads the M bits from its memory, performs an arbitrary computation on them, and sends its output to the machines that need it in the next round. After the final round of computation, the machines write their result to the final output location. If at any point one of the machines received more than M bits per round, the computation fails.

The model as described is quite general, we next specify the trade-offs between the size of the input (N), the number of machines (p), and the memory for each machine (M). Clearly in order to store the whole input for the first round, we must have $p \geq N/M$. The key limitation that makes the MPC model different from other systems is the upper bounds placed on both M and p. Specifically, we require that $M = N^{1-\epsilon}$ for some constant ϵ, thus the amount of input available to each machine on any round is a vanishingly small fraction (e.g., $M = \sqrt{N}$). The amount of parallelism is also restricted, and we require that $p = N^{1-\epsilon'}$ for some constant ϵ'. Clearly $\epsilon' + \epsilon \leq 1$, else all of the input cannot be stored.

The main metric for evaluating the efficiency of an algorithm in this model is the number of rounds that it takes. We aim for algorithms that take a constant number of rounds, but will see that sometimes that is not possible, and that a logarithmic number of rounds is necessary.

2.1.1 Discussion

While it may seem that requiring ϵ and ϵ' to be *arbitrary* constants gives too much freedom to the algorithm designer, currently almost all of the algorithms developed can be adapted to be used with arbitrary settings of ϵ, at the cost of increasing the number of rounds of computation within a constant factor. In fact, we will often see algorithms that run in $O(\frac{1}{\epsilon})$ rounds.

From the setting of the parameters, it is easy to see that the total size of the intermediate computation is bounded by $M \cdot p = N^{2-\epsilon-\epsilon'}$. Thus, it is beneficial to keep ϵ' as large as possible to limit the potential increase in intermediate computation size. While the model allows for ϵ' to be an arbitrarily small constant, we will again see that most algorithms are memory-efficient in their intermediate computation, and setting $\epsilon' \approx 1 - \epsilon$ is enough.

2.2 Example: Word Frequencies in Two Rounds

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

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

Round 1

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

Round 2

For each word w, the machine designated for it computes $\sum_{i=1}^{p} c_{w,i}$ and outputs the resulting count.

Analysis

The algorithm runs in two rounds and clearly computes the correct result. Initially, the data is partitioned equally across the machines so the memory per machine is $O(N/p) = O(\sqrt{N})$; thus the memory requirements are met with $\epsilon = 1/2$. Since there are $O(\sqrt{N})$ distinct words, each can be assigned a unique machine given the choice of p. Moreover, each machine sends and receives at most $O(\sqrt{N})$ pieces of information between the two rounds, satisfying the communication requirements.

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

2.2.1 Easy and Hard Problems in MPC

The above warm-up demonstrates the power and versatility of the MPC model. Similar techniques can be used to compute statistics about datasets (e.g., the minimum or maximum element, median, other quantiles) in constant number of rounds. These basic subroutines can then be adapted to use for other algorithmic primitives like sorting.

However, many problems that are easy in the sequential model of computation are much harder in MPC. As we will show in the rest of the sections, we need new methods to adapt other common paradigms such as dynamic programming, or common problems such as computing matchings in bipartite and general graphs.

For graphs, in particular, there is an important distinction between dense ($\Omega(n^{1+\epsilon})$ edges) and sparse ($O(n)$ edges) settings. In the latter, $M = N^{1-\epsilon} = o(n)$, and therefore one cannot load all of the nodes onto a single machine. This leads to additional complexity, as even simple aggregations across nodes take multiple rounds. This aspect is key to the lower bound construction in Section 8. On the other hand, in the dense case, it is permissible to keep all of the nodes (and even $\Omega(n)$)

edges on a machine, which makes some of these operations significantly easier and allows for additional techniques, such as Sample and Prune (Section 4) and Round Reduction (Section 6).

2.3 Other Related Models

2.3.1 Streaming

In the standard streaming model the input arrives in a sequence that can be read in only one direction; reading the whole input sequence from the first to the last constitutes a pass on in the input. The primary goal is to perform a certain computation with a small number of passes using memory whose size is sublinear in the input size. This model has some subtle differences from the MPC model. In the streaming model the machine is allowed to read the whole input, albeit in one direction. In contrast, in the MPC model no single machine ever sees the entire input. This makes it hard in general to simulate a streaming algorithm in MPC. Further, streaming algorithms are typically expected to use a much smaller memory (in many cases, $\operatorname{poly}\log N$) than what each machine is allowed in the MPC model; this makes it hard in general to simulate a MPC algorithm in streaming.

A variant of the streaming model is one augmented with a sorting primitive. In this variant, a machine is capable of annotating each item of the input with a key and sorting the items in the sequence according to the keys; this sorting step is considered a pass. Sorting greatly empowers the streaming model and in fact powerful enough for simulating MPC algorithms (albeit without the parallelism). For example, graph connectivity is a hard problem in the standard streaming model but can be solved in $O(\log N)$ passes when sorting is allowed.

Yet another variant of streaming that is closer to MPC is the MUD (massive, unordered, distributed) computational model. In this model the computation has three stages: a local function operates on each item of the input and produces a message; an aggregator that combines (pairs of) the messages; an optional final post-processing step. The output can depend on the order in which the aggregator operates on the messages. Streaming algorithms can easily simulate the MUD model while the converse, for general functions, is not true. For symmetric functions, however, an algorithm in the MUD model can simulate a streaming algorithm, with the same communication complexity and square of the space complexity.

2.3.2 Parallel Computing

NC and PRAMs

The standard parallel random access machines (PRAM) model allows the use of an arbitrary number of machines that share an unbounded memory in a synchronous way. In each round, each machine does certain local computations and reads from and/or writes to the shared memory. A primary goal for studying this model is to understand how much parallel computing can be exploited to minimize the number of rounds. NC defines the class of problems that be solved efficiently in the PRAM model. A problem is in NC if a polynomial (in N) number of machines can solve the problem in $\operatorname{poly}\log N$ rounds.

Although the PRAM has been the most popular model to understand the pure power and limitations of parallel computing, the model has seen limited application in practice. This is partially due to its lack of restrictions on the number of machines, (even a linear number is unrealistic when the input is large), as well as synchronization of such large number of machines. The MPC model is a more realistic alternative with fewer number of machines but each with significantly more computing power.

Although the two models, PRAM and MPC, are different, any CREW PRAM algorithm using at most $O(N^{2-\epsilon})$ machines and $O(N^{2-\epsilon})$ memory can be simulated in MPC Here, CREW stands for concurrent read and exclusive write, meaning that more than one machine can read a memory cell concurrently but only one can write to it in each round.

BSP

The bulk synchronous parallel model, aims to develop a more realistic parallel computing model. In this model, computation proceeds in a series of super-steps. In each super-step, each machine performs some local computations and sends messages to other machines. The machines synchronize between the super-steps.

The model has parameters for the memory needed per machine, local computation time, and time to send messages. The goal is to reduce the total computation time, which is the sum over the rounds of the of the maximum local computations and the maximum time for a machine to receive all the messages.

MPC is a special case of the BSP model where the memory in the machines is restricted, and the number of rounds is used to measure the running time. BSP gives a more refined analysis of distributed algorithms, whereas, MPC streamlines the model to allow the algorithm designer to focus on what are typically the most important aspects, for example, the number of rounds.

2.3.3 Distributed Computing

Local Model

In the Local model, there is an undirected graph where each node represents a machine. Each node, while unaware of the topology, can communicate only with its neighbors by sending any amount of data in each round. All communication happens synchronously in each round. Note that this model is not as restrictive on the amount of communication allowed per round as the MPC model, but the communication itself is restricted to between a node and its neighbors. At the end of the computation, each node is required to know its portion of the solution, e.g., for a coloring problem, each node should know its color in a proper coloring.

Despite the apparent differences, local algorithms have inspired new MPC algorithms for various graph algorithms, including maximal independent set, matching, and coloring; as well as reductions between the models to prove conditional lower bounds in MPC (see Section 8).

Congested Clique Model

The congested clique model studies the power and the limitations of distributed computing when the amount of data transferred between each pair of computationally unbounded machines is limited for each round, e.g., $O(\log N)$ bits. Note that each machine can communicate with all the other machines simultaneously; thus, each machine can receive $\Omega(N)$ bits in total from other machines in each round. This is unlike the MPC model where each machine in each round can send/receive $o(N)$ bits. On the other hand, the congested clique model is more restrictive than MPC as it limits the communication between each pair of machines. Although the two models are not directly comparable to each other, it is known that any MPC algorithm can be efficiently simulated in the congested clique model assuming that every machine has a sufficiently large memory to store all the data it receives from other machines in a round.

2.4 Section Notes

MapReduce, as a parallel computing system, stems from the seminal work of Dean and Ghemawat (2008). The first model formalizing MapReduce was given by Feldman *et al.* (2010), who gave more of a streaming variant. The MPC model, which is the focus of this book was introduced by Karloff *et al.* (2010).

The Bulk Synchronous Parallel (BSP) model was originally proposed by Valiant (1990), and its connections to MPC were explored by Goodrich (2010).

The closely related Local model for distributed computing was proposed by Linial (1987). While making different trade-offs than MPC, the Local model has inspired numerous MPC algorithms, and lower bound techniques.

Finally, the connections between MPC and the Congested Clique model were first explored by Hegeman and Pemmaraju (2015).

3

Partitioning and Coresets

The size of the problem instance is a critical constraint for MPC algorithms. Therefore, a natural approach is to try and shrink the given instance until it becomes small enough to fit in the memory of a single machine. A simple way to achieve such a size reduction is to partition the given instance into smaller sub-instances across the available machines, solve the sub-instance on each of the machines to obtain sub-solutions, and then combine the sub-solutions to obtain a solution to the given instance.

We call this the *partitioning* approach. When the problem at hand has no dependence between parts of the input, the partitioning approach is particularly powerful. For instance, suppose we want to count the number of times a specific word appears in a large corpus. It is easy to see that splitting the corpus across the machines into smaller sub-corpora, counting the number of occurrences of the word in each sub-corpus, and combining all the counts yields the correct solution, while achieving the instance-size reduction we seek.

Perhaps what is more surprising is that the same high-level partitioning approach works in situations where seemingly there is a lot dependence between different parts of the input in producing an optimum solution.

3.1 Overview

Problems for which the partitioning approach can be applied have the following characteristic. Let f be the function that we wish to compute and let $\mathcal{A}$ be an algorithm that computes or approximates f. Let X be the given input. We define the following property of $\mathcal{A}$ that will be useful in designing algorithms based on partitioning.

Definition 3.1 (Partition-friendly Algorithm)**.** An algorithm $\mathcal{A}$ is *partition-friendly* if for any input X and for any partition $X_1 \cup \cdots \cup X_p$ of X, it holds that $\mathcal{A}(\cup_{i=1}^{p} \mathcal{A}(X_p))$ computes or approximates $f(X)$.

Consider the following MPC algorithm to compute or approximate f using a partition-friendly algorithm $\mathcal{A}$.

Algorithm 1 Partitioning

```
PARTITION_A(X)
Arbitrarily partition $X$ equally into $X_1, \ldots, X_p$
For the $i$th partition $X_i$, let $Y_i = \mathcal{A}(X_i)$
Fetch $Y_i$'s from the partitions to one machine to form $Y = \cup_{i=1}^{p} Y_i$
return $\mathcal{A}(Y)$
```

Since $\mathcal{A}$ is partition-friendly, the solution output by Algorithm 1 is an approximation to f. Now we discuss the memory requirements of the algorithm. Since X was partitioned equally, the following statement is easy to see.

Lemma 3.1. Let r be the size of the output of $\mathcal{A}$. Then, setting the number of partitions $p = \sqrt{|X|/r}$ ensures both $|Y|$ and each of the $|X_i|$'s are of size $O(\sqrt{|X| \cdot r})$.

3.2 Application: Minimum Spanning Tree

Consider the classical *Minimum Spanning Tree (MST)* problem. Let $G = (V, E)$ be an undirected graph on $|V| = n$ vertices and $|E| = m$ edges, and let $w_e \in \mathbb{R}^+$ be the weight of edge $e \in E$. For ease, let the edges have distinct weights.

Problem 3.2.1 (Minimum Spanning Tree). Given $G = (V, E)$, find a subset $T \subseteq E$ of edges such that the edge-induced subgraph on V is connected, and the total weight of T, $\sum_{e \in T} w_e$, is minimized.

To apply the partitioning framework, for simplicity, we define $\mathcal{A}$ to be any algorithm to find the minimum spanning *forest*. This is so that we can use it for graphs that are not connected. Let the input X be the set E of edges, along with their weights. Let T be the output of Algorithm 1. We need two facts about $\mathcal{A}$. The first is that the since it outputs a forest, trivially the size of the output is n. Since $|X| = m$, by Lemma 3.1, setting $p = \sqrt{m/n}$ will make each sub-instance to be of size $O(\sqrt{mn})$, a substantial reduction from the size of the input graph G when m is considerably greater than n.

Second, we need to argue that $\mathcal{A}$ is partition-friendly. In fact, we will show that T is indeed the MST of G.

Lemma 3.2. T is the MST of G.

Proof. Let $E_i = X_i$ be the partition of E and let $T_i = Y_i$ in Algorithm 1.

Our proof proceeds by showing that none of the edges that was excluded in any partition, i.e., the edges in $E_i \setminus T_i$, can be a part of the MST of G. Note that since the edge weights are distinct, the MST is unique.

Recall the cycle property of MSTs: the maximum weight edge in any cycle in G cannot be in the MST, and conversely, any edge not in the MST is the heaviest edge on some cycle in G.

Now consider an edge $e \in E_i \setminus T_i$ that was removed by the algorithm. By the cycle property, e must be the heaviest edge on some cycle $C \subset E_i$. Since $E_i \subset E$, the edge is also the maximum cost edge on the same cycle C in G, and therefore is not part of the optimal MST T. □

Note that the approach above requires partitions be of size at least $\Omega(n)$. If the system constraints are such that one cannot afford to store all of the graph vertices on a single machine, then the partition approach no longer works and we need to use other methods to find the MST in parallel.

3.3 Application: k-Center Clustering

The *k-center clustering problem* is one of the classical clustering problems. At a high level, given a set of n points, the goal is to cover them all by k balls of minimum radius. Formally, consider a metric space equipped with a distance function $d(\cdot,\cdot)$ and let X be a set of points in it. The metric assumption implies that (1) $d(x,y)) = d(y,x)$ for all $x, y \in X$, (2) $d(x,x) = 0$ and (3: triangle inequality) $d(x,y) + d(y,z) \geq d(x,z)$ for all $x, y, z \in X$. We assume each point takes constant memory to store and the distances $d(x,y)$ can be computed in constant time given x and y. For a set C in the metric space and a point $x \in X$, let $d(x,C) = \min_{c \in C} d(x,c)$.

Problem 3.3.1 (k-Center)**.** Given $X = \{x_1, \ldots, x_n\}$, select k points $C = \{c_1, \ldots, c_k\}$ to minimize $\max_{x \in X} d(x,C)$.

The k-center problem is NP-hard, however there is a simple greedy algorithm that produces a 2-approximation.Further, this approximation guarantee is best possible for general metric spaces.

The greedy algorithm begins by selecting an arbitrary point $x \in X$ as the first center c_1. It then repeatedly selects the point in X that is furthest away from any of the previously chosen centers, and adds it to the set of centers. The resulting algorithm is a 2-approximation to the k-center problem. See Algorithm 2 for a formal description.

Algorithm 2 Sequential greedy algorithm for k-center.

```
Seq-KC($X, k$)
$C \leftarrow \emptyset$
Add an arbitrary point $x \in X$ to $C$
while $|C| < k$ do
    $y \leftarrow \arg\max_{x \in X} \; d(x, C)$
    $C \leftarrow C \cup \{y\}$
end while
return $C$
```

We will once again appeal to the partitioning framework by defining $\mathcal{A}$ to be Seq-KC. Let X be the input set of points and C be the output of Algorithm 1. As before, we need two facts about $\mathcal{A}$. The first is the easy observation that the output is of size k. Setting $p = \sqrt{n/k}$, we can ensure no sub-instance is of size more than $\sqrt{nk}$, which is sublinear in the input size for $k = o(n)$.

Second, we show that $\mathcal{A}$ is partition-friendly, however with a slightly worse approximation factor than $\mathcal{A}$.

Lemma 3.3. Algorithm 1 with $\mathcal{A}$ being Seq-KC outputs C that is an 8-approximation to the k-center of X.

Proof. Let OPT denote the value of the optimum k-center solution. We first argue that for any i and any $x \in X_i$, it holds that $d(x, C_i) \leq 4\mathrm{OPT}$. Indeed, since $X_i \subset X$, the value of the optimum solution OPT_i on this partition is no larger than $2\mathrm{OPT}$. Since Seq-KC returns a 2-approximation, we have $d(x, C_i) \leq 2\mathrm{OPT}_i \leq 4\mathrm{OPT}$.

Now consider $Y = \cup_i C_i$. Since $Y \subset X$, as before, the value of the optimum solution OPT_f is at most the value of $2\mathrm{OPT}$. Therefore, for any $x \in Y$, we have $d(x, C) \leq 4\mathrm{OPT}$.

Putting these two facts together yields a 8-approximation. Indeed, consider any point $x \in X$. If $x \in C_i$ for some i, then $d(x, C) \leq 4\mathrm{OPT}$. Otherwise, $x \in X_i$ for some partition X_i with clustering C_i, and let $z \in C_i$ be the point minimizing $d(x, C_i)$. Then, by the triangle inequality,

$$d(x, C) \leq d(x, z) + d(z, C) \leq 4\mathrm{OPT} + 4\mathrm{OPT} = 8\mathrm{OPT}. \qquad \square$$

A more careful analysis can show an approximation ratio of 4.

3.4 Coresets

An interesting generalization of the partitioning framework is the composable coreset framework. In this framework the input is still partitioned into $X_1, \ldots, X_p$. However, instead of using the algorithm $\mathcal{A}$ twice as in Algorithm 1, the idea is to use two different algorithms, one on the partitions and another on the union of the outputs of the first algorithm. The first algorithm can be thought of as computing a good *summary* of the input partition. More formally,

Definition 3.2. An algorithm $\mathcal{A}'$ computes an α-composable coreset if for any input X and for any partition $X_1 \cup \cdots \cup X_p$ of X, there exists an algorithm $\mathcal{A}$ such that $\mathcal{A}(\cup_{i=1}^{p} \mathcal{A}'(X_i))$ computes an α-approximation of $f(X)$.

Intuitively this can be useful because in some settings when a local solution may not contain all the information needed to compute a good global solution. As an application of this more general framework, we will obtain an improved algorithm for k-center in Euclidean space.

An interesting variant of the α-composable coreset framework is the randomized version, in which the input partition is assumed to be random. Interestingly, there are problems (like submodular maximization and maximum matching) that do not admit a good composable coreset for an arbitrary input partition, but do admit for a random input partition.

3.5 Application: k-Center Clustering in Euclidean Space

We consider the k-center problem where the set X of input points is in the Euclidean space $\mathbb{R}^d$. In this setting, assuming d is constant, it is possible to improve upon the 8-approximation in Theorem 3.3 using the composable coreset framework. The main idea is to design an algorithm $\mathcal{A}'$ that outputs a set of points so that every point x in an input partition X_i will have a close neighbor in $\mathcal{A}'(X_i)$.

Toward this goal, on every partition X_i we run the sequential algorithm Seq-KC (Algorithm 2) to obtain centers $c_{i,1}, \ldots, c_{i,k}$ and their respective clusters $\mathcal{C}_{i,1}, \ldots, \mathcal{C}_{i,k}$. For a cluster $\mathcal{C}_{i,j}$ let $\delta_{i,j}$ be the maximum distance between a point in $\mathcal{C}_{i,j}$ and $c_{i,j}$. Then for every cluster $\mathcal{C}_{i,j}$ we construct a $\delta_{i,j} \times \cdots \times \delta_{i,j}$ d-dimensional grid centered at $c_{i,j}$ and given $\epsilon > 0$, we further partition this grid into $\frac{\epsilon}{16\sqrt{2d}}\delta_{i,j} \times \cdots \times \frac{\epsilon}{16\sqrt{2d}}\delta_{i,j}$ d-dimensional sub-grids. Now for every non empty sub-grid the algorithm $\mathcal{A}'$ chooses an arbitrary input point in X_i that falls inside the sub-grid; the output $\mathcal{A}'(X_i)$ is the set of all the points chosen from X_i.

To prove that the algorithm computes a useful composable coreset we need to show: (i) $Y = \cup_i \mathcal{A}'(X_i)$ is not too big and (ii) an algorithm $\mathcal{A}$ such that $\mathcal{A}(Y)$ is a good approximation to the Euclidean k-center. We start with observing the first property, which is easy to see.

Lemma 3.4. $|Y| = O\left(\frac{1}{\epsilon^d} kp\right)$, where p is the number of input partitions.

Lemma 3.5. There is an algorithm $\mathcal{A}$ such that $\mathcal{A}(Y)$ is a $(2+\epsilon)$-approximation to the k-center of X.

Proof. First note that each $\delta_{i,j}$ is upper-bounded by 2OPT. Indeed, since $X_i \subset X$, the value of the optimum solution on this partition OPT_i is no larger than OPT and moreover, SEQ-KC is a 2-approximation algorithm. So for every i and every $x \in X_i$ $d(x, \mathcal{A}'(X_i)) \leq \frac{\epsilon}{4}\mathrm{OPT}$.

Now consider any point $x \in X$. By construction, it has a point in Y at distance at most $\frac{\epsilon}{2}\mathrm{OPT}$. This implies that there is a solution of cost at most $\mathrm{OPT} + \frac{\epsilon}{2}\mathrm{OPT}$, by the triangle inequality. Then once again using the SEQ-KC on Y, we obtain a $(2+\epsilon)$-approximation. □

As before, using partitions of size $\sqrt{n/k}$ and $\sqrt{n/k}$ machines with memory $O\left(\frac{1}{\epsilon^d}\sqrt{nk}\right)$, we can obtain a $(2+\epsilon)$-approximation in two rounds.

3.6 Problems

1. Generalize the MST algorithm for finding the maximum weight independent set under a matroid constraint. More formally, consider a matroid of rank k where U is the set of elements and $\mathcal{I}$ is a collection of sets that are independent in the matroid. Say that each element in $u \in U$ has a positive weight w_u and the weight of set in $\mathcal{I}$ is the sum of the weights on the elements in the set. Give an algorithm that computes the maximum weight set in $\mathcal{I}$ in $O(1)$ MPC rounds.

2. Consider the maximal matching problem. We are given an undirected simple graph $G = (V, E)$. Let n denote the number of vertices and m the number of edges. For this problem we will assume that $m = \Theta(n^2)$. Thus, the input size is $N := \Theta(n^2)$. The goal is to select a set of edges $S \subset E$ that is a maximal[1] matching in $O(1)$ MPC rounds.

3. Count the number of triangles in a given graph in $O(1)$ rounds.

[1] A subset S of edges is a matching if no two edges in S share an end point. Further, a matching is maximal if there is no matching S' such that $S' \supsetneq S$.

3.7 Section Notes

Partitioning is a common algorithmic technique and it was used for MPC for the first time in Karloff *et al.* (2010). The generalization to composable coresets is due to Indyk *et al.* (2014)

Triangle counting via partitioning for MapReduce was introduced by Suri and Vassilvitskii (2011), see Park *et al.* (2014) for an extension to multiple rounds and Afrati *et al.* (2012) on a discussion on the size of the intermediate computation.

The 2-approximation algorithm for clustering is due to Gonzalez (1985). The $(2+\epsilon)$-approximation for k-center can be improved to 2 using Sample and Prune; see Section 4.3 and the Notes in Section 4 for further improvements.

4

Sample and Prune

In this section, we consider MPC algorithms based on a specific type of sampling and testing framework. Generally, sampling is a powerful tool to quickly estimate statistics of large data sets. For instance, uniform sampling works for simple problems, such as computing the average: choosing a uniform random subset of the input and computing its average is likely to be a good estimate of the true average. However, for more complex problems, different elements have different influence on the final solution, and uniform random sampling may miss these important elements and result in a significantly suboptimal solution.

In this section we present a powerful framework that we term *sample and prune*. This is an iterative sampling method used to hone in on important elements in the input, precisely addressing the previous issue.

4.1 Overview

Problems that are amenable in the sample-and-prune framework ask to find a subset of size at most k from a universe U of size n optimizing a given objective, subject to some constraints. When parallelism is not required, these problems can usually be solved to (near-) optimality by a simple greedy algorithm $\mathcal{G}$. For instance, $\mathcal{G}$ may be the greedy algorithm to find a maximal independent set: given a list of vertices, iteratively add a vertex to the solution as long as it is not adjacent to any of the previously added vertices. Generally, we assume $\mathcal{G}$ is a set function that can be applied to any subset of U and returns a set of size at most k. Further restrictions on $\mathcal{G}$ will be discussed shortly. Recall that our memory size is $\tilde{\Theta}(M)$.

Algorithm 3 Sample-and-prune framework.

```
SAMPLE-AND-PRUNE$_{\mathcal{G}}(U)$
$R \leftarrow U$
$S \leftarrow \emptyset$
while $|R| > M$ do
    Sample each element in $R$ independently with probability
    $\tilde{\Theta}(M/|R|)$ and add it to $S$
    Construct a tester $\mathcal{T}_S^{\mathcal{G}} : R \to \{\texttt{KEEP}, \texttt{PRUNE}\}$ using $\mathcal{G}$,
    where $\mathcal{T}_S^{\mathcal{G}}(e) = \texttt{PRUNE}$ if and only if $\mathcal{G}(S) = \mathcal{G}(S \cup \{e\})$.
    $R \leftarrow R \setminus \{e \in R \mid \mathcal{T}_S^{\mathcal{G}}(e) = \texttt{PRUNE}\}$
end while
return $\mathcal{G}(R \cup S)$
```

The framework presented in Algorithm 3 repeatedly samples input elements to form a subset S, and then uses a tester $\mathcal{T}_S^{\mathcal{G}}$ based on the algorithm $\mathcal{G}$ to discard many of the input elements, thereby reducing the problem size. More formally, the algorithm maintains the invariant that $R \cup S$ contains enough information to find a good solution to the problem. In particular, it ensures that the optimum solution to the union of the sample S and the remaining elements R remains a good approximation to the solution to the original input U. The main algorithmic challenge in this framework is to design an efficient tester $\mathcal{T}_S^{\mathcal{G}}$ that satisfies the following two properties: (i) It should not prune any important elements still remaining in R; and (ii) After each sample, it should help to prune out a large number of elements.

Before we delve into the analysis of the sample-and-prune framework, we first discuss how to implement it in MPC by outlining how to run each of the iteration steps in Algorithm 3. The sampling step can be done in parallel if the elements are evenly partitioned across the machines. The samples from the machines can be brought together on one machine, where the tester $\mathcal{T}_S^{\mathcal{G}}$ can be constructed. The tester is then communicated to every machine and pruning step can once again be done in parallel. Each of the above takes one MPC round.

We now proceed with the analysis, beginning with two definitions.

Definition 4.1 (Usefulness). Fix $S \subseteq U$ and a set function $\mathcal{G}$. An element $e \in U \setminus S$ is *useful* for S under $\mathcal{G}$ if $\mathcal{G}(S) \neq \mathcal{G}(S \cup \{e\})$; otherwise, it is *useless.*

We also need the following monotonicity property, satisfied by most greedy algorithms.

Definition 4.2 (Monotonicity). A set function $\mathcal{G}$ is said to be *monotone* if for any $S \subseteq U$, if e is useless for S, then it is useless for any $S' \supset S$.

A natural way of constructing a tester $\mathcal{T}_S^{\mathcal{G}}$ satisfying Property (i) is to check for each $e \in R$ if e is useful for S, i.e., check $\mathcal{G}(S)$ against $\mathcal{G}(S \cup \{e\})$. If e is useless for S, then it can be pruned; $\mathcal{T}_S^{\mathcal{G}}(e)$ returns PRUNE. Otherwise, $\mathcal{T}_S^{\mathcal{G}}(e)$ returns KEEP.

We first argue the correctness of Algorithm 3.

Lemma 4.1. If $\mathcal{G}$ is monotone, the solution $\mathcal{G}(R \cup S)$ returned by Algorithm 3 is as good as the solution returned by $\mathcal{G}(U)$.

Proof. If an element e is pruned by $\mathcal{T}_S^{\mathcal{G}}$ during an iteration of the algorithm, then it is useless for S, which by monotonicity of $\mathcal{G}$ implies e is useless for U as well. In other words, no useful element was pruned and hence $\mathcal{G}(R \cup S) = \mathcal{G}(U)$. □

The only remaining thing to show is that at each iteration, a large number of (useless) elements are pruned. To do this, we first upper bound the size of the family of testers for a monotone $\mathcal{G}$.

Lemma 4.2. For any $S, |S| > k$, we have $\mathcal{T}_S^{\mathcal{G}} \equiv \mathcal{T}_{\mathcal{G}(S)}^{\mathcal{G}}$. Hence, the total number of distinct testers is $O(n^k)$.

Proof. Since $\mathcal{G}$ is monotone, if e is useless for $\mathcal{G}(S)$, then it will be useless for $S \supseteq \mathcal{G}(S)$. Therefore, every possible output of $\mathcal{G}(S)$ corresponds to a possibly unique tester. Since $\mathcal{G}$'s output has at most k elements, the total number of distinct testers is at most $\sum_{i \le k} \binom{n}{i} = O(n^k)$. $\square$

We now define what it means for a tester in this family to be good. Consider an iteration of Algorithm 3 and let R be the remaining set of elements during the iteration and S be the set of elements sampled so far.

Definition 4.3 (Good tester). A tester family $\{\mathcal{T}_S^{\mathcal{G}}\}_S$ is *good* if for all $S \subseteq U$ it satisfies the following property: $\mathcal{T}_S^{\mathcal{G}}(e) = \texttt{PRUNE} \iff \mathcal{T}_S^{\mathcal{G}} \equiv \mathcal{T}_{S \cup \{e\}}^{\mathcal{G}}$, for all $e \in U$.

In other words, if a tester $\mathcal{T}_S^{\mathcal{G}}$ decides to prune an element $e \in R$, then it will be functionally identical to that of a tester constructed by adding e to S. At the same time, if $\mathcal{T}_S^{\mathcal{G}}$ decides to keep an element $e \in R$, then it will differ from the tester constructed by adding e to S, i.e., e has a material effect on the behavior of the tester.

For simplicity let $M = n^\delta$ for some constant $0 < \delta < 1$. In each iteration, let the sampling probability be $1/(n^{1-\delta})$. Then, the number of samples in each iteration is $O(n^\delta)$ in expectation and is $O(kn^\delta \log n)$ with probability $1 - n^{-k}$ by a tail bound. We now discuss the effectiveness of the test in pruning out useless elements. Fix an iteration. We wish to argue that a lot of elements are pruned by the tester. By contradiction, fix a set R that is large, specifically, $|R| > 10kn^{1-\delta} \log n$.

Recall that R is the set of elements that were kept (not pruned). We claim that sampling any element from R would have changed the tester. This follows from the definition of a good tester. Indeed, for all S and e we have $\mathcal{T}_S^{\mathcal{G}}(e) = \texttt{PRUNE} \iff \mathcal{T}_S^{\mathcal{G}} \equiv \mathcal{T}_{S \cup \{e\}}^{\mathcal{G}}$ and therefore for any $e' \in R$, $\mathcal{T}_S^{\mathcal{G}}(e') = \texttt{KEEP}$ and $\mathcal{T}_S^{\mathcal{G}} \not\equiv \mathcal{T}_{S \cup \{e'\}}^{\mathcal{G}}$. Therefore, for a fixed tester, we conclude that no element from R could have been sampled.

We can bound the probability of having the set R by the probability of not sampling every element in R. Missing all elements in R happens with probability

$$\left(1 - \frac{1}{n^{1-\delta}}\right)^{|R|} \le \exp\left(-10kn^{1-\delta} \log n \cdot \frac{1}{n^{1-\delta}}\right) = n^{-10k}.$$

Next, we would like to union bound this probability over all possible large R. Naively done, this will be futile since there are $\binom{n}{10kn^{1-\delta}\log n}$ different R's. Instead, the key observation is that the total number of testers is relatively small, i.e., $O(n^k)$ by Lemma 4.2. Thus, using the fact that the tester (and the fixed S) determines R, we can simply apply a union bound over the number of testers.

This gives the following strategy to solve a problem with sample-and-prune. Find a monotone greedy algorithm $\mathcal{G}$ and a corresponding good tester. If the greedy algorithm returns at most k elements then the above analysis bounds the number of rounds.

4.2 Application: Top k Selection

As a starting example to illustrate how the sample-and-prune framework works, we consider the problem of finding the largest k out of the given n elements (say, numbers); this is the *top k selection* problem. For simplicity, let us assume every element is distinct. Note that the algorithm we present is not the most memory-efficient for this problem.

An obvious way of selecting the top k elements is repeatedly choosing the largest element from the remaining pool; call this greedy algorithm $\mathcal{G}$. It is easy to see that $\mathcal{G}$ is monotone and returns at most k elements.

The tester $\mathcal{T}_S$ in this case is easily realized based on $\mathcal{G}$: if e is not in the top k elements in $S \cup \{e\}$, then it outputs `PRUNE`. Indeed, if e is not in $\mathcal{G}(S \cup \{e\})$ then $e \notin \mathcal{G}(S' \cup \{e\})$ for $S' \supseteq S$. Thus, $\mathcal{T}_S$ is a good tester. The framework in Section 4.1 ensures the algorithm runs in a small number of rounds.

We now argue the algorithm returns the optimal solution. We know the tester will never discard one of the top k elements. Thus, they must be returned in the end.

4.3 Application: k-Center Clustering

Recall the k-center clustering problem (Problem 3.3.1). We will give a 2-approximation solution in this section using sample-and-prune. For simplicity, we assume that the value of the optimum objective, OPT, is known to the algorithm. We can always run the algorithm for each guessed value of OPT in parallel, with only a logarithmic factor blowup in the memory requirements, by sacrificing a constant factor in the approximation to OPT.

Our algorithm is based on one of the simple 2-approximate sequential greedy algorithms $\mathcal{G}$ that works as follows. Set $B = \emptyset$. Fix an ordering of the elements of U. Add the first point in U to B, and remove all points from U that are within distance 2OPT from B. Repeat this until $U = \emptyset$. Output B as the final solution. To see the algorithm obtains a 2-approximation, it suffices to show that it terminates after selecting at most k points.

We will build our tester on this greedy algorithm. Let $\mathcal{G}(S)$ be the output $\mathcal{G}$ gives on the sampled input S. Intuitively, we will discard each point $u \in U$ if it is within distance 2OPT from $\mathcal{G}(S)$. The only tricky part is that since the greedy algorithm considers input points in order, and its output is sensitive to the order the elements are presented. To handle this, we will construct a canonical ordering on the fly over iterations of the Algorithm 3. Let S_i be the points sampled in iteration $i \geq 1$ and let P_i be the points pruned in the same iteration.

We will inductively define a canonical ordering of the points. For a base case let $C_1 = \mathcal{G}(S_1)$. Construct a partial order where all points in C_1 come before $S_1 \setminus C_1$. Inductively, let $C_i = \mathcal{G}(S_i \cup_{j=1}^{i-1} C_j) \setminus C_{i-1}$ where $\mathcal{G}$ consider points in $\cup_{j=1}^{i-1} C_j$ first and then considers other points of S_i in an arbitrary but fixed order. This will ensure $\mathcal{G}$ returns $\cup_{j=1}^{i-1} C_j$ first. The tester $\mathcal{T}^{\mathcal{G}}_{S_1 \cup \cdots \cup S_i}$ in the ith iteration prunes a point if it is within distance 2OPT from a point in $C_1 \cup \cdots \cup C_i$. The canonical ordering of U is $C_1, S_1 \setminus C_1, P_1$, followed by $C_2, S_2 \setminus C_2, P_2, \ldots$. Note that the tester $\mathcal{T}^{\mathcal{G}}_{S_1 \cup \cdots \cup S_i}$ is equivalent to running the greedy on the points $C_1 \cup P_1 \cup \cdots \cup C_{i-1} \cup P_{i-1} \cup C_i$ in the canonical order and pruning a point if and only if it discards the point.

Under this canonical ordering, $\mathcal{G}$'s monotonicity is immediate: if a point is within distance 2OPT from X, then it must be within distance 2OPT from X' for any $X \subseteq X' \subseteq U$. Next, the algorithm returns a 2-approximation since points are only discarded if they are within 2OPT of $C_1 \cup \cdots \cup C_\ell$ where ℓ is the last iteration of the algorithm.

4.4 Application: Monotone Submodular Maximization Subject to a Cardinality Constraint

We next consider the problem of maximizing a monotone submodular function under a cardinality constraint and provide an almost $\frac{1}{2}$-approximation using a simple threshold greedy algorithm. The input to this problem is a universe U of n elements and a parameter k. The goal is to select k elements from U to maximize an objective function f that is monotone and submodular. To streamline the discussion, we assume each element in U takes constant memory to store, the submodular function can be stored in constant memory and can be evaluated on a machine in constant time.

We begin by formally defining the type of objective function.

Definition 4.4 ((Monotone) Submodular function)**.** Let $f : 2^U \to \mathbb{R}^+$. Define $f_A(e) := f(A \cup \{e\}) - f(A)$ to be the *marginal increase* of f when e is added to a set A. The function f is said to be *submodular* if for every $S' \subseteq S \subseteq U$ and $u \in U \setminus S$, we have $f_{S'}(u) \geq f_S(u)$. A function f is said to be *monotone* if $S' \subseteq S \implies f(S') \leq f(S)$.

While there is a natural greedy algorithm that gives a $(1 - 1/e)$-approximation,to illustrate the sample-and-prune technique, we will use a simpler greedy algorithm $\mathcal{G}$ called threshold greedy, which achieves $\frac{1}{2}$-approximation. The algorithm $\mathcal{G}$ is based on the following observation.

Lemma 4.3. Let f be a monotone submodular function. If $S = \{s_1, \ldots, s_t\} \subseteq U, t \leq k$ is any set such that

(i) there is an ordering of elements in S such that for all $0 \leq i < t$, $f_{\{s_1,\ldots,s_i\}}(s_{i+1}) \geq \mathrm{OPT}/(2k)$ and

(ii) if $t < k$, then $f_S(u) \leq \mathrm{OPT}/(2k)$, for all $u \in U$.

Then, $f(S) \geq \mathrm{OPT}/2$.

Proof. Let OPT be the value of an optimal solution S^* and let $\tau = \mathrm{OPT}/(2k)$. If $t = k$, then $f(S) = \sum_{i=0}^{k-1} f'_{S_i}(s_{i+1}) \geq \sum_{i=0}^{k-1} \tau = k\tau = \mathrm{OPT}/2$. If $t < k$, we know that $f'_S(u^*) \leq \tau$, for any element $u^* \in S^*$.

Furthermore, since f is submodular, $f(S \cup S^*) - f(S) \leq \tau \cdot |S^* \setminus S| \leq \mathrm{OPT}/2$. We have

$$f(S) \geq f(S \cup (S^* \setminus S)) - \frac{\mathrm{OPT}}{2} \geq \mathrm{OPT} - \frac{\mathrm{OPT}}{2} = \frac{1}{2}\mathrm{OPT},$$

using the monotonicity of f. □

This lemma can be used to obtain a greedy algorithm as follows. For simplicity, as in Section 4.3, we assume that the algorithm knows the value of OPT. If S in the lemma statement has k elements, then S is immediately $\frac{1}{2}$-approximation since every element in S has a marginal gain at least $\mathrm{OPT}/(2k)$. Otherwise, if every element not in S has a marginal gain of at most $\mathrm{OPT}/(2k)$ when added to S, the S is still a $\frac{1}{2}$-approximation. This gives us the greedy algorithm $\mathcal{G}$, which sequentially accepts an element if and only if the element yields a marginal gain of at least $\frac{1}{2k}\mathrm{OPT}$; up to k elements are accepted.

To build the tester using $\mathcal{G}$, we will inductively define an ordering of the elements as in Section 4.3. The tester and the canonical ordering are constructed on the fly.

We now argue that this is a good tester. Indeed, to see the tester is monotone, we just need to check that if $f_{E'}(e) < \frac{1}{2k}\mathrm{OPT}$, then $f_E(e) < \frac{1}{2k}\mathrm{OPT}$ for any $E \subseteq E' \subseteq U$. This is an immediate consequence of f's submodularity. As observed, the tester is designed to be equivalent to running the greedy on the elements in the canonical order. Thus, the algorithm is $\frac{1}{2}$-approximation by Lemma 4.3. The number of rounds follows from the general sample-and-prune framework.

4.5 Section Notes

This section is based on work in Kumar *et al.* (2015), Lattanzi *et al.* (2011), and Im and Moseley (2015). Nemhauser *et al.* (1978) showed that the natural greedy algorithm gives a $(1 - 1/e)$-approximation to the maximization of submodular functions in the sequential setting. Natural adaptions of the algorithm are asymptotically worse in the distributed setting; however, the breakthrough results of Ponte Barbosa *et al.* (2015) and Ponte Barbosa *et al.* (2016) showed how randomization coupled with the greedy algorithm can yield a $(1 - {}^{1}\!/_{e})$-approximation.

This can further be extended to show results when optimizing over various constraints such as a knapsack or matroid constraints.

The k-center problem has been considered using partitioning-based techniques in Malkomes *et al.* (2015) and Ceccarello *et al.* (2019).

5

Dynamic Programming

Dynamic programming (DP) is a powerful algorithm design technique and is part of the basic toolkit. DP typically consists of the following steps: (i) identifying subproblems, (ii) finding the right relationship between them, often in the form of a recurrence, and (iii) implementing the recurrence efficiently by reusing results from previous computations. A vanilla DP implementation that closely follows the recurrence is usually sequential, making it less appealing for the MPC model. In this section we study a framework to implement some DP algorithms in the MPC model. A central difficulty in defining such a framework is that the recurrences are often problem-dependent, making it harder to find a generic method. Nevertheless, we identify key properties that enable the existence of such framework in a principled way.

In particular, we use a framework that adapts a class of dynamic programs to the MPC model in a polylogarithmic number of rounds. While the resulting algorithms are not the best possible in terms of the number of rounds, this approach serves to illustrate the main ideas.

5.1 Overview

To introduce the framework we start by identifying key properties that allow for implementation in the MPC framework using a polylogarithmic number of rounds. These properties can be rather rigid, and we later discuss how to relax them via approximation; in particular we will show how to use approximate dynamic programs to obtain efficient algorithms.

Let us begin by defining some additional notation. Let $x_1, \ldots, x_n$ be the input; we assume that the input is ordered a priori. Now consider a dynamic program recurrence $D(i, v)$, where the first entry i is the index of the input element and the second entry $v \in V$ is a value; here V is the set of possible values. It is intuitive to think of v as a integer, but in some applications it could be a vector.

We now define two properties for $D(\cdot, \cdot)$ that will allow us to implement it efficiently in MPC.

Memory efficiency: Each entry of $D(i, v)$ should be take up polylogarithmic memory and can contain one of at most a polylogarithmic number of possible v values.

Bounded dependencies: Consider any two *consecutive* non-empty intervals I_1 and I_2 of the input such that $I_1 = [j, k]$ and $I_2 = [k + 1, \ell]$ with $0 \le j < k < \ell \le n$. Let $D_1(\cdot, \cdot)$ denote the *optimal* entries of the dynamic program if the entire input only consisted of I_1. Similarly let $D_2(\cdot, \cdot)$ denote the optimal entries of the dynamic program if the entire input only consisted of I_2. Suppose that we would like to compute the optimal entries of $D_{1\cup 2}(\cdot, \cdot)$ corresponding to the interval $[j, \ell]$, i.e., the concatenation of I_1 and I_2. Then it should hold that: (i) For any $i \in I_1$, for any $v \in V$, $D_{1\cup 2}(i, v)$ can be computed using the entry $D_1(i, v)$, a polylogarithmic number of entries from D_2, $|I_1|$, and $|I_2|$. We will let $S_{1,2}(i, v)$ be the set of entries from D_2 needed to compute $D_{1\cup 2}(i, v)$. When the intervals I_1 and I_2 are clear in the context we may drop the subscripts $1, 2$ and $1 \cup 2$. (ii) The set $S_{1,2}(i, v)$ can be obtained using only $D_1(i, v)$, $|I_1|$, and $|I_2|$.

The two properties guarantee that we can efficiently compute $D(\cdot, \cdot)$ in MPC. We will formalize this next.

5.1.1 Round Efficiency

We now show how to compute $D(\cdot, \cdot)$ in logarithmic number of rounds, using the properties of memory efficiency and bounded dependencies.

We start by partitioning the input $x_1, \ldots, x_n$ across the machines, respecting the ordering. Let $X[k]$ denote the interval (i.e., subset) of indices of inputs stored on machine k. Each machine k computes $D(i, v)$ for each v and for each $i \in X[k]$.

Next machines start to communicate to each other to compute the optimal $D(\cdot, \cdot)$ for larger input sizes. Before describing the algorithm we introduce some additional notation. Let $X[k_1, k_2] = \cup_{k \in [k_1, k_2]} X[k]$, where $k_1 < k_2$. For brevity, let $X_i[k_1, k_2] := X[k_1 + k(i), k_2 + k(i)]$ where $k(i)$ is the first index of the machine storing input x_i. The computation occurs in iterations. In iteration ℓ, each machine computes $D(i, v)$ for element x_i in the machine, assuming that the only indices are $X_i[0, 2^\ell]$, we denote this quantity as $D_\ell(i, v)$.

To proceed, assume $D_{\ell-1}(i, v)$ has been computed considering the indices in $X_i[0, 2^{\ell-1}]$ and the goal now is for machine $k(i)$ to compute $D_\ell(i, v)$ considering the indices in $X_i[0, 2^\ell]$. By assumption of the bounded dependencies property, there is a set $S(i, v)$ of entries that can be computed only using $D_{\ell-1}(i, v)$. If the entries in $S(i, v)$ are known, then $D_\ell(i, v)$ can be optimally computed.

To do this, a fixed machine will have a collection of sets $S(i, v)$ of entries for each element x_i stored on the machine. Using these sets, in one round the algorithm collects the entries corresponding to $S(i, v)$ stored on other machines. In a second round, these machine communicate the entries back. Using them, the $D_\ell(i, v)$'s can be computed.

First we show that Algorithm 4 can be realized in the MPC model. The correctness of the algorithm directly follows from the assumptions.

Lemma 5.1. Algorithm 4 can be implemented in the MPC model with m machines in $O(\log m)$ rounds using $\tilde{O}(\frac{n}{m}|V|)$ memory per machine, where each machine communicates $\tilde{O}(\frac{n}{m}|V|)$ bits per iteration.

Proof. Note that each machine stores $O(n/m)$ inputs and thus storing the $D(i, v)$'s for each $v \in V$ and each i that is local to the machine takes at most polylogarithmic memory by memory efficiency property.

Algorithm 4 Dynamic Program

```
Each machine $k$ computes $D(i, v)$ for all $v \in V$ using only its subset
$X[k]$ of the inputs
$D_0(i, v) \leftarrow D(i, v)$
for $\ell = 0, 1, \ldots, \log m$ do
    {Do the following in parallel}
    for each input index $i$ on machine $k$ and value $v \in V$ do
        Compute $S(i, v)$ from $D_\ell(i, v)$
        Fetch $D_\ell(i', v')$ for all $(i', v') \in S(i, v)$ by requesting from the
        respective machines
        {This is done for all $i$ on machine $k$ simultaneously}
        Compute $D_{\ell+1}(i, v)$ using $D_\ell(i', v')$ for all $(i', v') \in S(i, v)$
    end for
end for
return $D_{\log m}(\cdot, \cdot)$
```

For each iteration ℓ, a machine requests entries $S(i, v)$ for all i on the machine, where by the bounded dependencies property, $|S(i, v)|$ is polylogarithmic.

The proof is complete by observing that each iteration can be done in constant number of rounds and there are $O(\log m)$ iterations. □

5.2 Warm-up: Knapsack

We first consider the knapsack problem to illustrate the framework. This problem is significantly easier than the interval selection problem we will subsequently discuss but it will serve as a warm-up example.

In the *knapsack* problem, we are given n items where each item $i \in [n]$ has size s_i and weight w_i. The goal is to maximize the total weight of items packed into a knapsack of capacity S. This problem is weakly NP-hard and there exist algorithms whose running time is polynomial in S and n, or in $\sum_i w_i$ and n. For any subset $X \subseteq [n]$, let $w(X) := \sum_{i \in X} w_i$ and $s(X) := \sum_{i \in X} s_i$. Also let $W := w([n])$, which we assume is polynomial in n for simplicity. Recall the following sequential dynamic program where we compute $D(i, v) := \min_{X \subseteq [i] : w(X) \geq v} s(X)$. In words, $D(i, v)$ refers to the smallest memory one has to use to achieve weight v by choosing some items from the subset $[i]$. The following recurrence is elementary:

$$D(i, v) := \begin{cases} \min\{D(i-1, v), w_i + D(i-1, \max\{v - w_i, 0\})\} & i \in [n], v \in [W] \\ 0 & i = 0 \text{ or } v = 0 \end{cases}$$

For simplicity suppose the items are stored in the following manner: the first n/m items are in machine 1, the next n/m items are in machine 2, and so forth.

The first issue in simulating the dynamic program in MPC is that there are too many entries to compute. Hence, we will compress them by only considering weight values v that are powers of $1 + \epsilon$, which will incur a $(1 + \epsilon)$-factor loss in the approximation ratio; let V be the set of weights considered. The second issue is that each machine stores only n/m items and a naive implementation would need n/m rounds.

To fix the second issue, we will have to use parallelization more aggressively. To this end, we define an extended notion of subproblems. For simplicity of notation, assume that every machine has only one item, i.e., $m = n$; removing this assumption is straightforward. Machine i will be responsible for computing $\{D(i, v)\}_{v \in V}$. Let $D_\ell(i, v) := \min_{X \subseteq [i] \setminus [i - 2^\ell] : w(X) \geq v} s(X)$; we let $[i - 2^\ell] = \emptyset$ if $i \leq 2^\ell$. In words, $D_\ell(i, v)$ is a local version of $D(i, v)$ where it only pretends that there are items $i - 2^\ell + 1, \ldots i$. Clearly we have $D(i, v) = D_{\log n}(i, v)$.

Thus, to obtain an $O(\log n)$ round algorithm, it suffices to show how machine i can compute $D_\ell(i, v)$ in round ℓ, assuming that $D_{\ell-1}(\cdot, \cdot)$ has already been computed:

$$D_\ell(i, v) = \min_{v_1, v_2 \in V : v_1 + v_2 \approx v} D_{\ell-1}(i - 2^{\ell-1}, v_1) + D_{\ell-1}(i, v_2). \qquad (5.1)$$

Here, $v_1 + v_2 \approx v$ implies an approximate equality within a factor of $1 + \epsilon$, i.e., $v/(1 + \epsilon) < v_1 + v_2 \leq v$. Eqn. (5.1) can be easily shown by first assuming that v_1 and v_2 can take any integer value up to W and lowering their value as little as possible to ensure $v_1, v_2 \in V$. This approximation incurs a $(1 + \epsilon)$-factor loss in the value we obtain in every iteration, thus $(1 + \epsilon)^{\log n}$-factor in total. Therefore to obtain a $(1 + \epsilon)$-approximation we scale down ϵ by a factor of $\log n$. Thus, we will have $|V| = O(\frac{1}{\epsilon} \log W) = O(\frac{1}{\epsilon} \log n)$.

Note that this can be efficiently simulated in MPC because machine i can compute its own entries $\{D_\ell(i,v)\}_{v\in V}$ by accessing only the $O(\frac{1}{\epsilon}\log n)$ entries stored on machine $i-2^{\ell-1}$ (and itself).

5.3 Interval Selection in MPC

In this section, we show how interval selection fits into the MPC framework using Algorithm 4. For this problem we will introduce an approximate dynamic program. Later we will abstract out the approximate dynamic programming piece and discuss general guidelines for developing an approximate dynamic program in Section 5.4.

In the *interval selection problem*, there is a collection of intervals $x_1,\ldots,x_n$ where the ith interval x_i is $[s_i,e_i]$ with weight w_i. We assume intervals are *ordered by their ending time.* The goal is to select a subset of intervals of maximum total weight that do not intersect. To solve this problem, we consider the following dynamic program. Let $B(i,j)=v$ for some value $v\in[1,\sum_{i\in[n]}w_i]$ denote the optimal objective if the input instance only contains intervals $x_i,\ldots x_j$.

Unfortunately, the dynamic program $B(i,j)$ satisfies neither the memory efficiency property nor the bounded dependency property. The key idea is to transform this dynamic program into another with these properties.

The first idea is to "swap" the value v with the index j. Define a new recurrence $C(i,v):=\min\{j\mid B(i,j)=v\}$. That is, $C(i,v)$ is the least index j such that there is a solution to the sub-instance $\{x_i,\ldots,x_j\}$ with value at least v. Define $C(i,v)=\infty$ if $\{j\mid B(i,j)=v\}=\emptyset$.

We can compute $C(i,v)$ using the following recurrence. For the base case, we have $C(i,0)=i$ and $C(n,v)=\infty$ for all v. Inductively, for $v>0$ we have:

$$C(i,v)=\min_{v_1,v_2\geq 0}\{j_2\mid v_1+v_2=v, j_1=C(i,v_1), j_2=C(j_1,v_2)\}. \quad (5.2)$$

Intuitively, we guess a "split" of v into v_1 and v_2 and concatenate solutions corresponding to picking up value at least v_1 and value at least v_2. If we compute $C(i,v)$ for all v, then this suffices to compute $B(i,j)$ for all j.

Note that $C(\cdot,\cdot)$ still does not satisfy the memory efficiency and bounded dependencies properties. Interestingly we can relax it using approximation to solve this issue.

5.3.1 Approximate DP for Interval Selection

We now show how to approximate $C(i, v)$ using less memory. The key idea is to "sketch" the values. Specifically we want to compute the recurrence only for v's of the form ρ^k for some non-negative integer k, where $\rho := (1 + \epsilon/t)$ and t is a parameter to be chosen later. Note that since $B(1, n)$ is poly(n), we only need to consider integer exponents $k \in O(\frac{t}{\epsilon} \log n)$; let $V := \{\rho^k\}$ be the set of values considered.

Let $C'(i, v)$ be the approximation of $C(i, v)$ where $i \in [n]$ and $v \in V$. Note that $C'(\cdot,\cdot)$ has only $O(\frac{t}{\epsilon} n \log n)$ entries and can be computed as

$$C'(i, v) = \min_{v_1, v_2 \in V} \{j_2 \mid v/\rho \le v_1 + v_2 \le v \cdot \rho, j_1 = C'(i, v_1), j_2 = C'(j_1, v_2)\} . \tag{5.3}$$

Note that we lose a factor of $\rho = 1 + \epsilon/t$ in tracking the weights each time we apply this recurrence. If we think of this recursive computation as a tree, by choosing t to be its depth, we can still ensure an approximation of $\rho^t = 1 + O(\epsilon)$. Key is that the dynamic programming framework given for the MPC setting ensures the depth is at most $\log n$. Notice this does *not* depend on the values of the weights, but rather the total number of intervals.

Lemma 5.2. The recurrence $C'(\cdot,\cdot)$ satisfies the memory efficiency and bounded dependency properties.

Proof. We can assume that the weights are polynomial sized by sacrificing an arbitrarily small factory in the optimal value. For this reason, the recurrence is memory efficient as we need only consider a logarithmic number of entries ρ^k in V for $k \in [O(\log n)]$.

Now we consider the bounded dependency property. Consider any two substrings I_1 and I_2 of ordering such that $I_1 = [j, k]$ and $I_2 = [k + 1, \ell]$ with $0 \le j < k < \ell \le n$. Let $C'_1(i, v)$ be the recurrence in (5.3) computed on I_1 and similarly $C'_2(i, j)$ on I_2. Let $C_{1\cup 2}$ be the recurrence for $I_1 \cup I_2$. We can compute $C_{1\cup 2}(i, v)$ as follows from $C_1(\cdot,\cdot)$ and $C_2(\cdot,\cdot)$.

First $C_{1\cup 2}(i,v) = C_2(i,v)$ for all $i \in I_2$. Next consider an $i \in I_1$. If $C(i,v) \neq \infty$ we have $C_{1\cup 2}(i,v) = C_1(i,v)$. Otherwise we compute this as the following.

$$C'_{1\cup 2}(i,v) = \min_{v_1,v_2 \in V} \tag{5.4}$$

$$\{j_2 \mid v/\rho \leq v_1 + v_2 \leq v \cdot \rho, C'_1(i,v_1) \neq \infty, j_2 = C'_2(x_2,v_2) \tag{5.5}$$

$$\text{where } x_2 \text{ is the first element in } I_2 \text{ larger than } C'_1(i,v_1)\}.$$

In this recurrence we need only consider $|V|$ many entries from $C_2(\cdot,\cdot)$ for each i, v_1 pair. This is because there are V entries and each one has exactly one corresponding value v_2. The required indices (x_2, v_2) can be directly computed from $C_1(i,v_1)$. Thus, we have the bounded dependency property. □

This lemma along with Lemma 5.1 shows that the $C'(\cdot,\cdot)$ can be computed in MPC. In the next section, we bound guarantees on the quality of the solution returned by $C'(\cdot,\cdot)$.

5.3.2 Approximation Guarantees for Interval Selection

The next lemma bounds the final guarantee of computing $C'(\cdot)$ using Algorithm 4. The key is that in each recursive step a $(1+\epsilon/t)$ factor can be lost in the objective value. However, we know that the longest chain of dependencies in the dynamic programming computation is at most $O(\log n)$. This follows from Algorithm 4 running in $O(\log n)$ rounds and only on such dependency is added for any entry in each iteration.

Lemma 5.3. Computing C using Algorithm 4 returns a $(1-O(\epsilon))$-approximation to $B(1,n)$.

Proof. We show the following stronger claim: For an input x_i and value $v^* \in V$, let $C_\ell(i,v^*)$ be the value of $C(i,v^*)$ pretending that we only have the intervals in $X_i[1,2^\ell]$. Then, there exists $v \in V$ such that $v > v^*/\rho^\ell$ and $C'_\ell(1,v) \leq C_\ell(1,v^*)$. Note that this claim would complete the proof as the value of the solution given by the subproblems $C'_\ell(i,v)$ corresponds to the largest $v \in V$ such that $C'_\ell(i,v) < \infty$.

We prove the claim by induction on ℓ. The base case for $\ell = 0$ is clear since we set $C'_0(i, v) = C(i, v)$ using only the intervals stored in machine k.

To show the inductive step, fix an iteration $\ell \geq 1$, interval i, and value $v^* \in V$. Consider the set S of inputs achieving v^*, i.e., $S \subseteq X_i[1, 2^\ell]$ and the total value of the inputs in S is v^*. Note that if S is fully contained in $X_i[1, 2^{\ell-1}]$ or in $X_i[2^{\ell-1}+1, 2^\ell]$ we are done by the induction hypothesis. Otherwise, let $X_{j_1^*}$ be the largest index input in $S \cap X_i[1, 2^{\ell-1}]$. Let v_1^*, v_2^* be such that $j_1^* = C_{\ell-1}(i, v_1^*)$, $j_2^* = C_{\ell-1}(j_1^*, v_2^*)$, and $v^* = v_1^* + v_2^*$. Note that v_2^* can be computed from the intervals in $X_i[2^{\ell-1} + 1, 2^\ell]$.

By the induction hypothesis, there exist $v_1, v_2 \in V$ with $v_1 > v_1^*/\rho^{\ell-1}$ and $v_2 > v_2^*/\rho^{\ell-1}$ such that

$$C'_{\ell-1}(i, v_1) \leq C_{\ell-1}(i, v_1^*) \text{ and } C'_{\ell-1}(j_1, v_2) \leq C_{\ell-1}(j_1, v_2^*),$$

where $j_1 = C'_{\ell-1}(i, v_1)$; if $X_{j_1} \in X_i[1, 2^{\ell-1}]$, then let I_{j_1} be the smallest index input in $X_i[2^{\ell-1} + 1, 2^\ell]$. Due to the way the algorithm is defined, we have $C'_\ell(i, v) \leq C_{\ell-1}(j_1, v_2^*)$, where $v \in V$ such that $v > (v_1 + v_2)/\rho > (v_1^* + v_2^*)/\rho^\ell = v^*/\rho^\ell$. Note that $j_1 = C'_{\ell-1}(i, v_1) \leq C'_{\ell-1}(i, v_2^*) = j_1^*$ and both inputs X_{j_1} and $X_{j_1^*}$ are in $X_i[2^{\ell-1} + 1, 2^\ell]$. Further, recall that v_2^* is computed from the inputs in $X_i[2^{\ell-1} + 1, 2^\ell]$. Thus, by definition of $C'_{\ell-1}$, we have $C_{\ell-1}(j_1, v_2^*) \leq C_{\ell-1}(j_1^*, v_2^*)$, which immediately implies $C'_\ell(i, v) \leq C_{\ell-1}(j_1^*, v_2^*) = C_\ell(i, v^*)$, as desired. □

The above lemmas imply the following result, setting $t = \Omega(\log n)$.

Theorem 5.4. There is an MPC algorithm to compute a $(1 - \epsilon)$-approximate solution to interval selection. The algorithm uses $\tilde{O}(n/m)$ memory per machine and runs in $O(\log n)$ rounds.

5.4 Approximate Dynamic Programs

Many natural dynamic programs will not satisfy the bounded dependency property or memory efficiency. However, often it is possible to transform a dynamic program $D(i, v)$ for $i \in [n]$ and $v \in V$ into an alternative dynamic program $C(i, v)$ for $v \in V_C$ where V_C is much smaller that V so that $C(i, v)$ satisfies the bounded dependency and memory efficiency properties. The trade-off is that $C(i, v)$ is sub-optimal, but is an approximate solution.

An example of this was given in the prior section. In this section, we give general guidelines for reducing the dependencies via approximation.

Approximation via Sketching: Consider a recurrence $D(i, v)$ where $v \in V$ corresponds to the objective value. Say that there is a corresponding recurrence $C(i, v)$ where $v \in V_C$ such that $V_C \subseteq V$ and V_C has poly-logarithmic size so that $C(\cdot, \cdot)$ satisfies memory efficiency. Further, assume that C has a corresponding recurrence satisfies the bounded efficiency property.

The recurrence C need not be an exact dynamic program. Rather, the key is the following. Say that we compute C using Algorithm 4. Further, say that in each recursive computation of C, at most a $(1 + \epsilon/t)$ factor is lost for any $0 < \epsilon < 1$ and integer t. Then by setting $t = \Omega(\log n)$, the proof in Lemma 5.3 can be extended to show that $C(\cdot, \cdot)$ is a $(1 + \epsilon)$-approximation for the overall problem.

Using such a framework it is possible to obtain $(1+\epsilon)$-approximation algorithms using $O(\frac{1}{\epsilon\delta})$ rounds of MPC with $O(n)$ total memory and $O(n^{\delta})$ memory per machine for constants $\epsilon, \delta > 0$ for the problems of weighted interval selection, optimal binary search tree, and the longest increasing subsequence.

5.5 Section Notes

The framework described in this section is due to Im *et al.* (2017).

6

Round Reduction via Sampling

In this section we develop a family of techniques to reduce the number of rounds of MPC algorithms when working with large graphs.

The key idea is to identify properties (e.g., vertex degree) that are well preserved when working with a random subset of the data. As our primary focus is on graphs, such a random subset can be obtained by sampling either edges or vertices; these two approaches leads to two different guarantees. As a case study, we will consider the k-core decomposition problem but the same technique can be applied to several classical problems such as matching and maximum independent set.

The k-core problem can be solved sequentially by a simple iterative sequential algorithm. In this section we will show how to use edge sampling to modify this algorithm to obtain an $O(\log n)$-round algorithm that uses near-linear memory per machine. We will then extend the analysis via vertex partitioning to get an $O(\log \log n)$-round algorithm that also uses near-linear memory.

6.1 k-core Decomposition and a Sequential Algorithm

A wide range of data mining, machine learning, and social network analysis problems can be solved by identifying dense regions in large graphs and understanding the role of vertices in a density-based hierarchical decomposition of the graph. A commonly used technique for this task is the k-core decomposition: the k-core of a graph is a maximal subgraph where every vertex has induced degree at least k.

Formally, let $G = (V, E)$ be a graph with $|V| = n$ vertices and $|E| = m$ edges. For a vertex $v \in G$ we denote by $d(v)$ the degree of the vertex in G. If H is an induced subgraph of G, for any vertex $v \in H$ we denote by $d_H(v)$ the degree of v in H. A *k-core* is a maximal subgraph $H \subseteq G$ such that $\forall v \in H$ we have $d_H(v) \geq k$. For any k the k-core is *unique* and possibly disconnected. We say that a vertex v has *coreness* number k if it belongs to the k-core but not to the $(k+1)$-core. We denote the coreness number of vertex v in the graph G by $C_G(v)$; we may drop the subscript notation when the graph is clear from the context. We let $V_{\geq t} := \{v \in V \mid C_G(v) \geq t\}$ denote the set of vertices with coreness number at least t in the original graph G. We define $V_{\leq t}$ similarly.

We also define the *core-labeling* for a graph G as the labeling where every vertex v is labeled with its *coreness* number. It is worth noting that this labeling is *unique* and that it defines a natural hierarchical decomposition of G as shown in Figure 6.1. In the sequential setting a simple iterative greedy algorithm computes the k-core decomposition efficiently. In Algorithm 5 we provide the pseudocode for it.

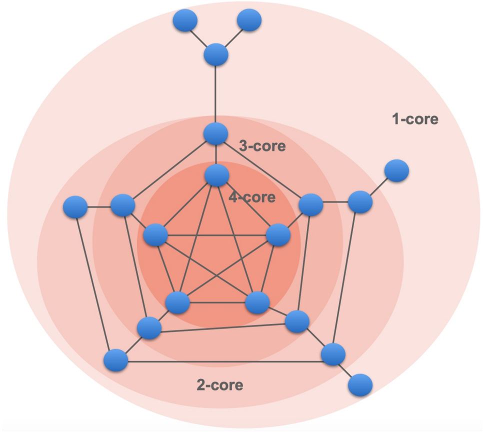

Figure 6.1: Hierarchical decomposition induced by the coreness numbers.

Algorithm 5 Sequential k-core algorithm.

```
Input: $G = (V, E)$
Output: Coreness numbers $\{C_G(v)\}_{v \in V}$
$V_H \leftarrow V$
for $\ell \leftarrow 0$ to $n$ do
    Repeatedly remove any $v$ with $d_H(v) = \ell$ from $V_H$ and set $C_G(v) \leftarrow \ell$
end for
```

6.2 Parallelizing the Sequential Algorithm: $O(\log n)$ Rounds

It is easy to construct a graph for which the sequential algorithm can take $\Omega(n)$ iterations to label all vertices. In this section we present a simple technique to obtain an MPC algorithm that only uses $O(\log n)$ rounds and almost linear memory to solve the problem approximately.

The first step toward this goal is to define relaxed notions of k-core decomposition and coreness number. We define an $(1-\epsilon)$-*approximate k-core* of G to be a subgraph H that (i) H contains the k-core of G and (ii) $\forall v \in H$, $d_H(v) \geq (1-\epsilon)k$. In other words, a $(1-\epsilon)$-approximate k-core of G is a subgraph of the $((1-\epsilon)k)$-core of G and a supergraph of the k-core of G. In Figure 6.2 we illustrate the 3-core and a 2/3-approximate 3-core for a small graph.

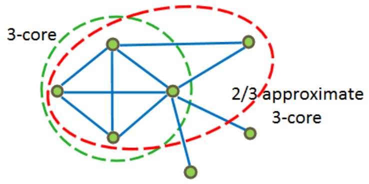

Figure 6.2: Example of 3-core and 2/3-approximate 3-core.

Similarly, in a $(1-\epsilon)$-*approximate core-labeling* of a graph G, each vertex is labeled with a number between its coreness number and its coreness number multiplied by $\frac{1}{1-\epsilon}$.

The main intuition behind our MPC algorithm is to compute coreness numbers in *decreasing* order by grouping them in $O(\log n)$ buckets and solving the problem for each bucket in $O(1)$ rounds. In particular for decreasing i, vertices with coreness numbers in the range $(2^{i-1}, 2^i]$ are processed, and all the edges whose both endpoints have coreness value larger than 2^i are ignored as they will already be labeled. The remaining edges are sampled with probability $\tilde{\Theta}(2^{-i})$, which reduces the maximum coreness value in the remaining graph to $\tilde{O}(1)$. Here, only the vertices whose degrees are considerably larger than 2^{i-1} still have incident sampled edges and can be labeled approximately. The reason starting from the bucket with the largest coreness numbers is because to fit the sampled edges into a single machine we have do significantly down sample edges, so only high-degree vertices are distinguishable after sampling.

The following pseudocode shows the $O(\log n)$-round MPC algorithm.

Algorithm 6 Approximate coreness for all vertices.

```
Input: $G = (V, E)$ and an approximation parameter $\epsilon$
Output: Approximate coreness number $\{\tilde{C_G}(v)\}_{v \in V}$
$\Gamma_0 \leftarrow \emptyset$
for $i \leftarrow 1$ to $\log_{1+\epsilon} n$ do
    Find $\Gamma_i \subseteq V$ s.t. (i) $\Gamma_i \supseteq V_{\geq (1-\epsilon)^{i-1} n}$ and (ii) $\Gamma_i \cap V_{\leq (1-\epsilon)^i n} = \emptyset$
    $\tilde{C_G}(v) \leftarrow (1-\epsilon)^{i-1} n$ for all $v \in \Gamma_i \setminus \Gamma_{i-1}$
end for
```

Recall that $V_{\geq t}$ denote the set of vertices with coreness number at least t in the original graph G. We label an unlabeled vertex v if its coreness number is too big for the iteration, i.e., $C_G(v) \geq (1-\epsilon)^{i-1}$; conversely, if a vertex's coreness number is too small, we defer its labeling to subsequent iterations. We have the freedom to handle the other vertices either way. It is immediate that this algorithm yields $(1-2\epsilon)$-approximate[1] coreness numbers in $O(\frac{1}{\epsilon} \log n)$ iterations.

It remains to show how to compute Γ_i in $O(1)$ rounds given Γ_{i-1} that satisfies the properties (i) and (ii). Towards this goal, we use the following memory-efficient algorithm based on edge sampling. It is essentially combining Algorithm 5 with edge sparsification. In the following note that H is *different* from the subgraph induced on $V \setminus \Gamma_{i-1}$.

[1] More precisely, $(1-\epsilon)^2$-approximate.

Algorithm 7 Computing Γ_i, given Γ_{i-1} satisfying (i) and (ii)

```
$k \leftarrow n(1-\epsilon)^{i-1}$ and $p \leftarrow \min\{1, \frac{50 \log n}{k\epsilon^2}\}$
$E_H \leftarrow$ edges sampled in parallel independently from $E \setminus E[\Gamma_{i-1}]$ with
    probability $p$, where $E[\Gamma_{i-1}]$ is the edges with both end points in $\Gamma_{i-1}$
Let $H = (V_H \leftarrow V, E_H)$
Repeatedly remove vertices $v$ from $V_H$ such that $d_H(v) < (1-\epsilon/2)pk$
$\Gamma_i \leftarrow V_H$
```

Lemma 6.1. Given Γ_{i-1}, Algorithm 7 correctly computes Γ_i, w.h.p.

Proof. We assume that $p < 1$, and consequently $50\frac{\log n}{k\epsilon^2} < 1$, as otherwise H equals G for the unlabeled vertices and the lemma immediately follows. Observe that this assumption also implies $k > 50\frac{\log n}{\epsilon^2}$, which will be important for showing that our bounds hold w.h.p.

We first show $V_{\geq k} \subseteq \Gamma_i$. By definition of coreness, each $v \in V_{\geq k}$ has at least k neighbors in $V_{\geq k}$. For any $u \in V_{\geq k}$, we thus have $\mathbb{E}[d_{H_0}(u)] \geq pk \geq 50\frac{\log n}{\epsilon^2}$, where H_0 is the subgraph H right after the sampling before no vertices are removed. Then, by a Chernoff bound, with probability at least $1 - n^{-4}$, $d_{H_0}(u) \geq (1-\epsilon/2)pk$. By a union bound, this holds for all vertices with probability at least $1 - n^{-3}$. Therefore, Algorithm 7 never removes vertices from $V_{\geq k}$, meaning $V_{\geq k} \subseteq \Gamma_i$.

We next show $\Gamma_i \cap V_{\leq (1-\epsilon)k} = \emptyset$. For each $v \in V_{\leq(1-\epsilon)k}$, let E_v denote the set of edges incident to v in the k'-core of G where $k' = C_G(v)$. In other words, E_v is the set of witness edges that certify that v has coreness number k'. By definition of $V_{\leq(1-\epsilon)k}$, $|E_v| \leq (1-\epsilon)k$. It then follows that $|E_v \cap H_0| \leq (1-\epsilon/2)pk$ for all $v \in V_{\leq(1-\epsilon)k}$ w.h.p. Thus, all vertices in $V_{\leq(1-\epsilon)k}$ are removed from V_H. □

Finally we argue that the algorithm can be realized using $\tilde{O}(n)$ memory per machine. This follows from the following simple observations.

Lemma 6.2. For any k, the number of edges incident to $V_{\leq k}$ is at most $k|V_{\leq k}|$.

Lemma 6.3. The number of edges sampled is $O(\frac{1}{\epsilon^2} n \log n)$ w.h.p.

6.3 Round Compression via Random Vertex Partitioning: $O(\log\log n)$ Rounds

In this section we show how to exponentially reduce the number of rounds used by the algorithm in the previous section. The main idea, instead of sampling the edges independently, is to randomly partition the vertices among different machines and analyze the induced subgraph in parallel. This "round compression" idea enables to simulate multiple rounds of the MPC algorithm in much fewer number of rounds.

Partitioning the vertices across $1/p$ machines uniformly at random is equivalent, from each machine's point of view, to sampling each vertex with probability $1/p$. The main advantage of vertex sampling over edge sampling, which was used in Section 6.2, is the following. A vertex of original degree d has an expected degree of pd after both edge sampling and vertex sampling (conditioned on it being sampled in case of vertex sampling). However, the resulting graph size is very different: the graph has mp edges in expectation after edge sampling but only mp^2 edges in expectation after vertex sampling. See Figure 6.3.

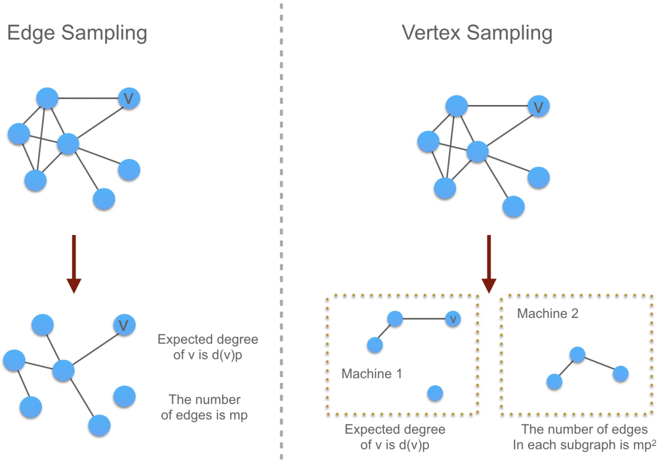
Edge Sampling
Vertex Sampling
V
V
V
V
Expected degree of v is d(v)p
The number of edges is mp
Machine 1
Machine 2
Expected degree of v is d(v)p
The number of edges In each subgraph is mp²

Figure 6.3: Difference between vertex sampling and edge sampling with $p = \frac{1}{2}$.

In Algorithm 6, we first sampled edges with probability $\tilde{\Theta}(1/n)$ (hiding logarithmic terms and setting $\epsilon = 1/2$) to identify high-degree vertices of degree in $[n/2, n]$. If the graph is dense, the resulting graph could have $\Theta(n)$ edges, which would barely fit in a single machine of memory $\tilde{\Theta}(n)$. This was the main bottleneck to parallelization. That is, identifying lower degree vertices requires more aggressive sampling. For example, to identify vertices of degree about $\sqrt{n}$, we have to sample edges with probability $p = \Theta(1/\sqrt{n})$. But the sampled edges can be as many as $mp = \Omega(n^{1.5})$ (for dense graphs). In contrast, vertex sampling with the same probability yields a subgraph of size $mp^2 = O(n)$. Thus, we can handle all degrees and eventually coreness numbers in $[\sqrt{n}, n]$ in $O(1)$ rounds. By subsequently handling vertices of degree $[n^{1/4}, n^{1/2}]$ $[n^{1/8}, n^{1/4}]$, $\ldots$, we will arrive at $O(\log\log n)$ rounds.

The following pseudocode is presented to formally describes the high-level flow, which is a natural generalization of Algorithm 6. The definition of Γ_i remains unchanged.

Algorithm 8 Approximate coreness via round compression.

```
Input: $G = (V, E)$ and an approximation parameter $\epsilon$
Output: Approximate coreness number $\{\tilde{C}_G(v)\}_{v \in V}$
$\Gamma_0 \leftarrow \emptyset$
for $i \leftarrow 1$ to $O(\log\log n)$ do
    $k \leftarrow n^{1/2^{i-1}}$
    for all $j$ such that $(1-\epsilon)^j n \in [\sqrt{k}, k]$ do
        Find $\Gamma_j \subseteq V$ satisfying (i) and (ii) described in Algorithm 6
        $C_G(v) \leftarrow (1-\epsilon)^{j-1} n$ for all $v \in \Gamma_j \setminus \Gamma_{j-1}$
    end for
end for
```

To obtain an $O(\log\log n)$-round algorithm, we would like to execute each iteration of the outer **for** loop in constant number of rounds. Formally, we show the following.

Lemma 6.4. Given a set Γ such that $V_{\geq k} \subseteq \Gamma$ and $V_{\leq (1-\epsilon)k} \cap \Gamma = \emptyset$, we can compute in $O(1)$ rounds Γ_j such that $V_{\geq (1-\epsilon)^{j-1} n} \subseteq \Gamma_j$ and $V_{\leq (1-\epsilon)^j n} \cap \Gamma_j = \emptyset$, for all for j such that $(1-\epsilon)^j n \in [\sqrt{k}, k]$.

The proof is almost as for Lemma 6.1.

Intuitively, this can be done in parallel because the same Γ is used for all such j. The following procedure adapts Algorithm 7 to use vertex sampling. It outlines how to compute each Γ_j.

Algorithm 9 Computing Γ_i for each j s.t. $(1-\epsilon)^{j-1}n \in [\sqrt{k}, k]$ given Γ.

```
$k' \leftarrow (1-\epsilon)^{j-1}n$ and $p \leftarrow \frac{50 \log n}{k' \epsilon^2}$
Assign each vertex $V \setminus \Gamma$ uniformly at random among $\frac{1}{p}$ machines
for each machine do
    Let $H = (V_H, E_H)$ be the induced subgraph on the vertices assigned to the machine
    Repeatedly remove vertices $v$ from $V_H$ such that $d_H(v) < (1-\epsilon/2)pk'$
    $\Gamma_j \leftarrow V_H$
end for
```

Lemma 6.5. The induced graphs H_i have $O(n \log^2 n/\epsilon^4)$ edges w.h.p.

Proof. Consider a fixed machine and consider $k' = \sqrt{k}$, which results in the highest sampling probability. Then, the fixed machine has an edge after the partitioning with probability $p^2 = \Theta(\frac{\log^2 n}{k\epsilon^4})$. Observation 6.2 implies that there are at most kn edges in $E[V \setminus \Gamma]$. Thus, the machine is assigned $O(\frac{\log^2 n}{k\epsilon^4})kn$ edges in expectation. An appropriate use of the Chernoff bounds completes the proof. □

6.4 Section Notes

The content of this section is mainly based on Esfandiari *et al.* (2018) and Ghaffari *et al.* (2019b) The $O(\log n)$-round algorithm presented in this section is inspired by the filtering technique from Section 4 and was first presented in Esfandiari *et al.* (2018). The use of round compression to efficiently estimate the coreness number was introduced in Ghaffari *et al.* (2019b).

Round compression has found many uses in the MPC model. The technique was first introduced by Czumaj *et al.* (2018), who obtain the first algorithm for matching that runs in sub-logarithmic number of rounds and uses near-linear memory. Their result has then been improved in a series of follow-up (Behnezhad *et al.*, 2019b; Ghaffari *et al.*, 2018; Assadi *et al.*, 2019a), leading to the current best result of a $(1+\epsilon)$-approximate matching in $O(\log \log n)$ rounds. These techniques have been used in related problems such as computing independent sets

(Ghaffari *et al.*, 2018), vertex cover (Assadi *et al.*, 2019a), and graph coloring (Chang *et al.*, 2019).

7

Round Reduction via Graph Exponentiation

In this section we introduce a new technique called graph exponentiation that improves the running time of MPC graph algorithms when memory per machine is strongly sublinear in the number of vertices. The main idea behind graph exponentiation is that if the decision of an algorithm for a vertex v in round i only depends on v's ith neighborhood[1], then we can simulate multiple rounds in parallel by loading the extended neighborhood for every vertex. Here is an example.

Suppose that an algorithm runs in $O(\log n)$ rounds and all the choices of the algorithm for every vertex in round i only depend on its ith neighborhood. Furthermore, suppose that an $O(\sqrt{\log n})$-neighborhood of every single vertex can fit on a machine. Then one could split the $O(\log n)$ rounds of the algorithm in $O(\sqrt{\log n})$ phases, each consisting of $O(\sqrt{\log n})$ rounds, and simulate the $O(\sqrt{\log n})$ rounds in a single round by aggregating the $O(\sqrt{\log n})$ neighborhood of each vertex into a single machine. It is possible to collect the $O(\sqrt{\log n})$ neighborhood of each vertex using only $O(\log \log n)$ rounds (by iteratively sending, in step j, the 2^{j-1}-neighborhood to all of the 2^{j-1}th neighbors). In

[1] The ith neighborhood of vertex v is defined as the set of vertices that are reachable from v by traversing at most i edges.

this way one would obtain an MPC algorithm using $\tilde{O}(\sqrt{\log n})$ rounds. Unfortunately in many cases the $O(\sqrt{\log n})$-neighborhood does not fit effortlessly in a single machine; hence, this technique is often combined with sampling and other problem-specific ideas. Figure 7.1 illustrates graph exponentiation.

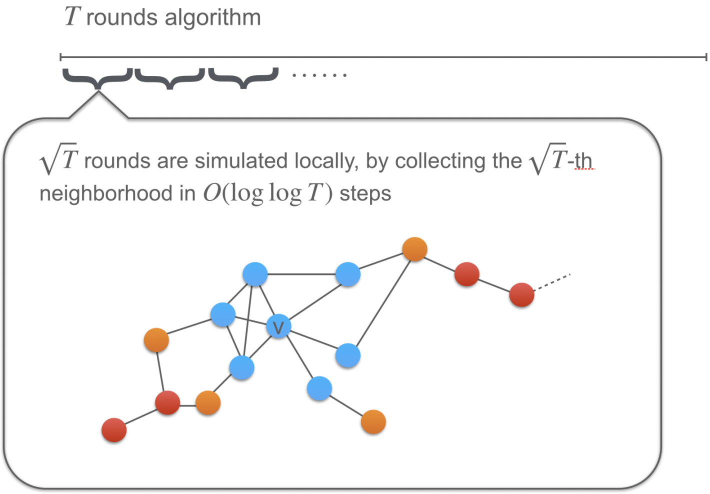
T rounds algorithm
......
$\sqrt{T}$ rounds are simulated locally, by collecting the $\sqrt{T}$-th neighborhood in $O(\log\log T)$ steps
v

Figure 7.1: Visual representation of the graph exponentiation technique. A T-round algorithm is divided in $\sqrt{T}$ phases of $\sqrt{T}$ rounds that are executed locally. In the figure, $T = 4$. To load the 4-neighborhood efficiently for every vertex in step j every vertex sends its 2^{j-1} neighbors to all its 2^{j-1}th neighbors. The vertices v receive in this doubling process in the first and second steps are colored orange and red, respectively.

We apply graph exponentiation approach to two problems, finding approximate core decomposition and finding connected components.

7.1 Approximate Core Decomposition

In Section 6 we presented efficient algorithms for approximate k-core decomposition that used memory per machine that is almost linear in the number of vertices of the graph. In this section we focus on the more challenging setting where machines have strictly sublinear memory, i.e., each machine has memory n^α for some constant $\alpha < 1$. In this setting, we first present a simple $(2+\epsilon)$-approximation algorithm that runs in $O(\log n)$ rounds; we then show, using graph exponentiation, how to improve it to run in $\tilde{O}(\sqrt{\log n})$ rounds.

7.1.1 $O(\log n)$-round Algorithm

For a graph $G = (V, E)$, let $V_{\le k}$ be the set of vertices with coreness number $\le k$ and let $D_{\le(2+\epsilon)k}$ the set of vertices with degree $\le (2+\epsilon)k$. We let $\mathtt{deg}_S(u)$ denote u's degree in the subgraph induced on $S \subseteq V$.

The $O(\log n)$-round algorithm is based on the following simple observation:

Lemma 7.1. $|V_{\le k} \setminus D_{\le(2+\epsilon)k}| \le \frac{|V_{\le k}|}{1+\epsilon/2}$.

Using this, for any fixed k we can find a $(2+\epsilon)$-approximate $V_{\le k}$ by the following method (Algorithm 10). The algorithm repeatedly removes vertices of degree less than $2(1+\epsilon)k$ and adds them to Λ. From Lemma 7.1 it follows that Λ is unchanged after $\Theta(\frac{1}{\epsilon}\log n)$ rounds.

Algorithm 10 Computing $(2+\epsilon)$-approximate $V_{\le k}$ for fixed k.

```
Input: Graph $G = (V, E)$ and a threshold $k$
Output: $\Lambda$ such that $V_{\le k} \subseteq \Lambda \subseteq V_{\le 2(1+\epsilon)k}$
$\Lambda \leftarrow \emptyset$
for $\Theta(\frac{1}{\epsilon}\log n)$ rounds do
    Find $S := \{u \in V \mid \mathtt{deg}_V(u) \le (2+\epsilon)k\}$
    $V \leftarrow V \setminus S$
    $\Lambda \leftarrow \Lambda \cup S$
end for
```

It is easy to see that Algorithm 10 computes Λ as desired. To implement the algorithm in the MPC model, we use the fact that it only needs to scan the adjacency list of each vertex to find S, which can be done using memory $\max(\Delta, n^\alpha)$, where Δ is the maximum degree in the graph. With slightly more care, the dependence on Δ can in fact be removed.

We next show how to compute approximate coreness numbers. For each integer $i \ge 1$, let Λ_i be such that $V_{\le(1+\epsilon)^i n} \subseteq \Lambda_i \subseteq V_{\le(2+\epsilon)(1+\epsilon)^i n}$ and let $\Lambda_0 := V_{\le 0}$. We can compute all Λ_i, $0 \le i \le \log\log_{1+\epsilon} n$ in parallel in $O(\frac{1}{\epsilon}\log n)$ rounds, using Algorithm 10. We declare u has coreness number $(2+\epsilon)\cdot(1+\epsilon)^i$ for the highest i such that $u \in \Lambda_i$. Then, from by definition of Λ_i, we have $(1+\epsilon)^{i-1}n \leq C_G(u) \leq (2+\epsilon)(1+\epsilon)^i n$, which implies a $((2+\epsilon)(1+\epsilon))$-approximate coreness labeling.

To summarize, a primitive that we used to obtain a $(2+\epsilon)$-approximate algorithm was the following: identifying and removing a set S of vertices in constant number of rounds from $V_{\leq k}$ such that (i) each vertex in S has coreness number at most $(2+\epsilon)k$ and (ii) the size of $V_{\leq k}$ decreases by a constant factor. Lemma 7.1 offers the basis for the primitive, which states S can be the set of vertices of degree at most $(2+\epsilon)k$ in the remaining graph.

7.1.2 $O(\sqrt{\log n})$-round Algorithm

To reduce the required number of rounds to $\tilde{O}(\sqrt{\log n})$, we will modify the primitive as follows: Can we remove a set S of vertices in $O(\log\log n)$ rounds from $V_{\leq k}$ ensuring (i) each vertex in S has coreness number at most $(2+3\epsilon)k$ and (ii) the size of $V_{\leq k}$ decreases by a factor of $1/(1+\epsilon/6)^{\sqrt{\log n}}$? If this is feasible, by repeatedly applying the primitive $\Theta(\sqrt{\log n})$ times, we will achieve $\tilde{O}(\sqrt{\log n})$ rounds. We will construct this new primitive will by compressing multiple rounds using graph exponentiation combined with a couple of other ideas, detailed below.

We start by observing a certain local property in Algorithm 10: whether a vertex u is removed from V in the ith iteration of the **for** loop depends only on u's ith neighborhood, i.e., the vertices that are within i hops from u. More precisely, whether $u \in S_i$ or not only depends on u's ith neighborhood. Indeed, this can be verified by seeing that every vertex u in S_i must be adjacent to some vertex in S_{i-1} for otherwise, u must have been added to the solution in the $(i-1)$st iteration.

As discussed earlier, this suggests a way to simulate Algorithm 10 in MPC more efficiently: if we could store the $O(\sqrt{\log n})$-neighborhood of every single vertex in a machine we could simulate $O(\sqrt{\log n})$ rounds in almost a single round; this would lead to an $\tilde{O}(\sqrt{\log n})$-round algorithm. Unfortunately it is often not possible to load the entire $O(\sqrt{\log n})$-neighborhood of every vertex in a machine. Thus, the key challenge lies in simulating graph exponentiation in a memory efficient manner.

The key idea is to use sampling. We observe that Lemma 7.1 offers some flexibility: instead of removing all vertices of degree less than $(2+\epsilon)k$, we can try to remove all vertices of degree, say, less than $(2+\epsilon/3)k$ but none of degree higher than $(2+3\epsilon)k$; the remaining vertices can be handled arbitrarily. It is easy to show that this modified process still decreases $V_{\leq k}$ by a constant factor. Thus, we only need to approximately measure the degree of each vertex.

The following lemma allows us to run Algorithm 10 using $(2+\frac{2\epsilon}{3})pk$ as a new threshold after sampling each edge with probability p. Note that (i) in the modified primitive comes from its guarantees. Its proof is a direct application of a Chernoff bound.

Lemma 7.2. Let G' be the graph obtained by sampling every edge of G with probability $p = \min\left\{1, \frac{10\log n}{(\epsilon/3)^2 k}\right\}$. Then, w.h.p. it holds:

- If $d_G(v) \leq (2+\frac{\epsilon}{3})k$, then $d_{G'}(v) < (2+\frac{2\epsilon}{3})pk$.
- If $d_G(v) \geq (2+3\epsilon)k$, then $d_{G'}(v) \geq (2+\frac{2\epsilon}{3})pk$.

However, even after sampling, the $O(\sqrt{\log n})$-neighborhood of some vertex may be too large to fit in $O(n^\alpha)$ memory. To tackle this problem, at the beginning of every phase we *freeze* vertices with significantly high degrees and we do not delete any frozen vertices in the phase. Interestingly we can show that this does not effect the algorithm too much. Indeed by Lemma 6.2, there are not many vertices with coreness number smaller than k yet have such high degree. Furthermore, by "ignoring" all the frozen vertices, after sampling the $O(\sqrt{\log n})$-neighborhood of every vertex fits in a single machine's memory, whp.

To make the high-level ideas more transparent, we only focus on how we can simulate the first $\sqrt{\log n}$ iterations of Algorithm 10 in $O(\log\log n)$ MPC rounds. We can make the total memory usage at most n^δ factor more than the whole input size for any constant $\delta > 0$. This is achieved by the following algorithm.

Every vertex in $V \setminus F$ has a degree at most h. Therefore, after sampling it has a degree at most hp in expectation and at most $2hp$ whp by a Chernoff bound. Thus, for every vertex $u \in V \setminus F$, u's $\sqrt{\log n}$-neighborhood in $(V \setminus F, E')$ has size at most $(2hp)^{\sqrt{\log n}} \leq n^{2\delta}$ for any constant ϵ and sufficiently large n. This implies that the $\sqrt{\log n}$-neighborhood fits in a single machine and we can process all non-frozen vertices in parallel by using at most $n^{2\delta}$ factor more memory than the input size in total. Further, as observed before, we can determine if Algorithm 10 adds v to Γ within $\sqrt{n}$ iterations only using v's $\sqrt{n}$-neighborhood. Using graph exponentiation, this can be simulated in $\log(\sqrt{\log n}) = O(\log \log n)$ rounds.

We now check properties (i) and (ii) needed for the primitive. Thanks to Lemma 7.2, every vertex added to Λ in Algorithm 11 has coreness number at most $(2+3\epsilon)k$; thus Λ satisfies (i). Further, due to Lemmas 7.2 and 7.1 with threshold $(2+\epsilon/3)k$, we know that $|V_{\leq k} \setminus F \setminus \Lambda| \leq \frac{1}{(1+\epsilon/6)^{\sqrt{\log n}}}|V_{\leq k} \setminus F|$. Further, by Lemma 6.2, we know that $|F| \leq 2k|V_{\leq k}|/h = \frac{1}{2^{\sqrt{\log n}}}|V_{\leq k}|$. Therefore, Λ satisfies (ii).

Algorithm 11 Compressing $\sqrt{\log n}$ rounds into $O(\log \log n)$-rounds via graph exponentiation

```
Input: Graph $G = (V, E)$ and a coreness threshold $k$
Output: $\Lambda$ such that (i) $\Lambda \subseteq V_{\leq (2+3\epsilon)k}$, (ii) $|V_{\leq k} \setminus \Lambda| \leq \frac{2}{(1+\epsilon/6)^{\sqrt{\log n}}}|V_{\leq k}|$
$F \leftarrow$ vertices with degree $\geq h := 2k \cdot 2^{\sqrt{\delta \log n}}$ (frozen vertices)
$E' \leftarrow$ edges sampled i.i.d. with probability $p = \min\left\{1, \frac{10 \log n}{(\epsilon/3)^2 k}\right\}$
Each vertex collects in $V \setminus F$ its $\sqrt{\log n}$-neighborhood in $(V \setminus F, E')$ in
    parallel in $O(\log \log n)$ rounds
Simulate Algorithm 10 with graph $(V \setminus F, E')$ threshold $(2 + \frac{2\epsilon}{3})pk$ for
    each $v \in V \setminus F$; if $v \in S_i$ for some $i \leq \sqrt{\log n}$, then add $v$ to $\Lambda$
```

Thus, by repeating Algorithm 11 q times such that $\left(\frac{2}{(1+\epsilon/6)^{\sqrt{\log n}}}\right)^q < \frac{1}{n}$, we will have Λ such that $V_{\leq k} \subseteq \Lambda \subseteq V_{\leq (2+3\epsilon)k}$. And this can be done in $O(\log \log n)q = \tilde{O}(\sqrt{\log n})$ rounds for any fixed constant $\epsilon > 0$. This produces essentially the same output as Algorithm 10 but using only $\tilde{O}(\sqrt{\log n})$ rounds. The remaining procedure is almost identical. Thus, we can compute $(2 + \epsilon)$-approximate coreness in $\tilde{O}(\sqrt{\log n})$ rounds.

7.2 Connected Components

Finding connected components in an unweighted graph is a fundamental primitive with many practical applications. In this section we first describe a basic sequential algorithm. Then we will discuss how to parallelize it to obtain an $O(\log n)$-round algorithm using sublinear memory. We will then show how to modify it using graph exponentiation to obtain an $\tilde{O}(\log D)$ algorithm, where D is the diameter of the graph.

We first present a simple sequential algorithm for finding connected components (Algorithm 12).

Algorithm 12 Sequential connected component algorithm.

```
**Input:** Graph $G = (V, E)$
**Output:** Set $\mathcal{S}$ of connected components of $G$
$\mathcal{S} = \emptyset$
$\forall v \in V, \mathcal{S} = \mathcal{S} \cup \{v\}$
**while** $\exists S \in \mathcal{S}$ that has an edge to another component $S' \in \mathcal{S}$ **do**
    $\mathcal{S} = \mathcal{S} \setminus S, S'$
    $\mathcal{S} = \mathcal{S} \cup \{S \cup S'\}$
**end while**
Output $\mathcal{S}$
```

Note that the algorithm can be efficiently implemented in sequential setting using a union-find data structure and it returns the correct connected components of the graph. Next, we will discuss how to parallelize this algorithm to run in $O(\log n)$ rounds.

7.2.1 $O(\log n)$-round Algorithm

The main idea to parallelize Algorithm 12 is to activate multiple connected components in every round. But, we cannot arbitrarily let every connected component to merge with a neighboring connected component at the same time, since it can lead to long merging chains.

To circumvent this problem we design a simple merging strategy where in every round the algorithm merges only disjoint sets of connected components. In particular, the following steps are done in constant number of rounds: (i) each connected component in $\mathcal{S}$ picks either red or green uniformly at random, (ii) each connected component informs all its neighboring connected components about its color choice, and (iii) every red connected component selects a random green neighboring connected component (if any). In this way disjoint "stars" are formed, each consisting of a central green connected component connected to one or more red neighboring connected component(s). Then, we merge all connected components in each star. Figure 7.2 presents a visual representation of the process. See Algorithm 13 for a formal description.

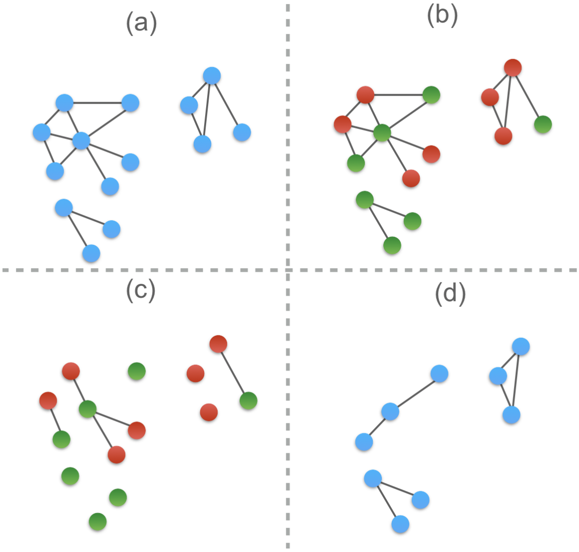

Figure 7.2: In step (a), we represent each connected component with a vertex. In step (b), each connected component picks a random color. In step (c) every red components pick a random neighboring green components and finally in step (d) the connected components are collapsed.

Theorem 7.3. For large enough constant C, Algorithm 13 computes with high probability the connected components of G in $O(\log n)$ rounds in MPC using machines with sublinear memory.

Proof. Let n^δ be the memory per machine where $\delta \in (\frac{1}{2}, 1)$[2]. We first discuss how to implement the algorithm in MPC and then prove its correctness.

[2] We assume that $\delta > \frac{1}{2}$ only for simplicity.

Algorithm 13 Parallel connected components algorithm.

```
Input: Graph $G = (V, E)$
Output: Set $\mathcal{S}$ of connected components of $G$
$\mathcal{S} = \emptyset$
$\forall v \in V, \mathcal{S} = \mathcal{S} \cup \{v\}$
for $C \log n$ rounds in parallel do
    for $S \in \mathcal{S}$ do
        $S$ selects a random color {red, green}
        $S$ sends its color to its neighboring connected components
        if $S$ is red and has at least one neighboring green connected component then
            $S$ select a random neighboring green connected component and merges with it
        end if
    end for
end for
Output $\mathcal{S}$
```

There are two obstacles to implementing Algorithm 13 in the MPC model. First, a connected component may have size more than n^δ, in which case it will be split across machines. Second, in the selection step, a green component may be selected by more than n^δ red components, in which case, again, the merge step needs to be distributed.

Both of these issues can be addressed by a single level of indirection: one can use a two level tree to route all of the necessary communication. For instance, in the first case, each large component can have n^δ vertices selected as "coordinators," and all of the other vertices are assigned to a unique coordinator. All of the communication is first passed to the coordinators, who then pass the information to the individual vertices. Since we had assumed $\delta > 1/2$, a single level of indirection suffices, but it is easy to extend the same argument for arbitrarily small δ. A similar approach can be used to handle the case of many red vertices connecting to a single green vertex.

Now we turn our attention to correctness. Clearly the algorithm does not merge disconnected components so we only need to show that every connected component is correctly identified. To prove this we focus on a single connected component C of size k. Initially, we have k

distinct components and we will now show that if we have more than one active connected component in C after each execution of the **for** loop, the number of connected components decreases in expectation by a factor of $\frac{3}{4}$. Note that a connected component is not merged in a round of the algorithm if it either selects the color green (happens with probability $\frac{1}{2}$) or if none of its neighbors is colored green (happens with probability at most $\frac{1}{4}$ since every component in C has at least one neighboring component.) So every component is not merged with probability at most $\frac{3}{4}$. Thus in expectation the number of components in C decreases by $\frac{3}{4}$ in every round. Hence with constant probability, a constant fraction of the connected components is merged in every round and so with high probability the process converges to a single connected component in $O(\log k)$ rounds. □

7.2.2 $\tilde{O}(\log D)$-round Algorithm

To improve the round efficiency of the algorithm presented in Section 7.2.1, a key observation is that it does not always make sense to sample the green and the red color with the same probability. Suppose hypothetically that all of the connected components would have neighborhood of size n^γ for constant γ in every round of the algorithm. Then, by coloring every component green with probability $n^{-\frac{\gamma}{2}}$ and red otherwise we would make more progress. In fact, the probability of a connected component failing to merge would be at most $n^{-\frac{\gamma}{3}}$ and so the algorithm would converge in $O(\frac{1}{\gamma})$ rounds. Unfortunately, the hypothetical condition may not hold. Nevertheless, we will show how to leverage on the double exponential size reduction techniques to circumvent this obstacle. In particular we show how to obtain an $O(\frac{1}{\gamma}\log D)$-round algorithm.

The main idea is that connected components with small neighborhoods can be efficiently modified to become connected components with large neighborhoods. In particular, every connected component can send all its neighboring components to all its neighbors iteratively. In this way, in i rounds every connected component receives its 2^i-neighborhood. Unfortunately one cannot run this exponentiation until convergence since it can require too much memory. However, one can run this process until every connected component has at least $n^{\frac{\delta}{2}}$ neighboring connected components. At that point every connected component colors itself green with probability $n^{-\frac{\delta}{3}}$ or red otherwise. Then every red connected component selects a neighboring green connected component and merges with it as before. In Figure 7.3 we present a visualization of the process, which can be shown to converge in $O(\frac{1}{\gamma}\log D)$ rounds.

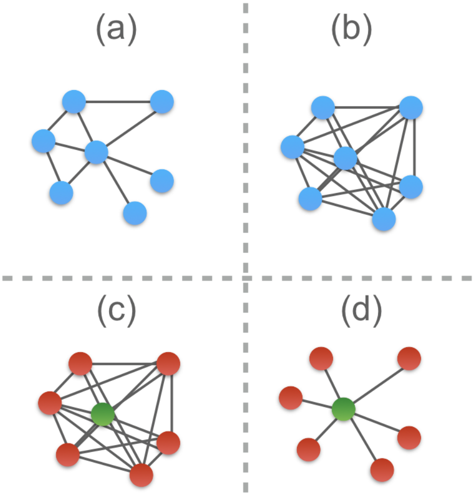

Figure 7.3: In step (a), we represent each connected component with a vertex. In step (b), grows its neighborhood. In step (c) every connected component select its colors randomly. In step (d) every red component picks a random neighboring green component and the connected components are collapsed.

We now present the pseudo-code in Algorithm 14.

Theorem 7.4. With high probability, Algorithm 14 computes the connected components of G in $O(\frac{1}{\delta}\log D)$-rounds in the MPC model where each machine has memory n^{δ}, $\frac{1}{2} < \delta < 1$.

Proof. The implementation details of this algorithm are very similar to that of Theorem 7.3. The only additional difficulty is to implementation the graph exponentiation process, but this can be easily done using a coordinator and by noticing that the total number of edges received by a single machine in the doubling phase is bounded by $O(n^{\delta})$.

Algorithm 14 $\tilde{O}(\log D)$ connected component algorithm.

```
Input: Graph $G = (V, E)$.
Output: The set $\mathcal{S}$ of connected components of $G$
$\mathcal{S} = \emptyset$
$\forall v \in V, \mathcal{S} = \mathcal{S} \cup \{v\}$
for $C \log n$ rounds in parallel do
    for $\log D$ rounds in parallel do
        for $S \in \mathcal{S}$ do
            if $S$ neighborhood is smaller than $n^{\frac{\delta}{2}}$ then
                if $S$ has a neighboring connected component $S'$ with $n^{\frac{\delta}{2}} + 1$
                neighboring connected component then
                    $S$ connects with all neighboring connected components of $S'$
                else
                    $S$ connects to all the neighboring connected components of
                    its neighboring connected component
                end if
            end if
        end for
    end for
    for $S \in \mathcal{S}$ do
        if $S$ neighborhood is larger than $n^{\frac{\delta}{2}}$ then
            $S$ colors itself *green* with probability $n^{-\frac{\delta}{3}}$ or red otherwise
            $S$ sends its color to its neighboring connected component
            if $S$ is red and has at least one neighboring green connected
            component then
                $S$ select a random neighboring green connected component and
                merge with it
            end if
        else
            if $S$ is the smallest id components between its neighbors then
                $S$ collapse all its neighbors in a single connected component
            end if
        end if
    end for
end for
Output $\mathcal{S}$
```

Furthermore, we note that in $O(\log D)$ rounds of graph exponentiation, a connected component is either transformed into a clique or into a graph where every vertex has degree at least $n^{\frac{\delta}{2}}$. In the first case, the connected component is found in a single round. In the second case, the number of active connected components decreases by a $n^{-\frac{\delta}{4}}$ factor, with high probability. Thus with constant probability the algorithm converges to the right solution in $O(\frac{1}{\delta}\log D)$ rounds. □

7.3 Section Notes

The content of this section is mainly based on Ghaffari *et al.* (2019b) and Andoni *et al.* (2018). In particular the first parallel algorithm using sublinear memory and sub-logarithmic rounds for k-core decomposition was presented in Ghaffari *et al.* (2019b). The first parallel algorithm using less then $O(\log n)$ parallel rounds and sublinear memory has been presented in Andoni *et al.* (2018).

The graph exponentiation technique was first introduced to speed-up algorithm in the distributed LOCAL model by Lenzen and Wattenhofer (2010) and then adapted to the MPC model by Ghaffari and Uitto (2019) to provide efficient algorithms for matching and maximum independent set. Algorithm 10 is reminiscent of the densest subgraph algorithm for MPC due to Bahmani *et al.* (2012a). To the best of our knowledge, there is no algorithm with better than 2-approximate algorithm that runs in $o(\log n)$ rounds using strictly sublinear memory for the k-core decomposition problem.

Developing algorithms for graph connectivity has been an important topic in the MPC model; see Andoni *et al.* (2018), Behnezhad *et al.* (2019a), and Coy and Czumaj (2022) for recent work on this topic.

8

Lower Bounds

Owing to numerous applications, graph problems have received significant attention in the MPC model. In previous sections, we studied MPC algorithms for vertex cover, connected components, maximum matching, independent set, and many others.

8.1 Connectivity in MPC

While new algorithms have been developed, there seems to exist a barrier to obtaining constant round algorithms for many of these problems. Intuitively, this may be due to graph properties depending on the global structure, which is difficult to grasp in a small number of rounds when each machine can only see part of the input graph. The sharpest example of this is the graph connectivity problem, for which we can state the following challenge:

> Given an n-vertex undirected graph whose edges are arbitrarily partitioned across machines each with memory $O(n^{\alpha})$ for some constant $\alpha < 1$, can we determine if the graph is connected in $o(\log n)$ rounds?

Recall the $O(\log n)$-round algorithm in Section 7.2. Can we do better?

To appreciate this question, we will consider the following problem.

Problem 8.1.1 (1-vs-2-cycle). Given an undirected graph on n vertices and n edges where the edges are arbitrarily placed across machines of memory size $M = O(n^{\alpha})$ for $\alpha < 1$, determine if it is a single cycle of length n or two cycles of length $n/2$.

Even this simple instance of the graph connectivity problem has resisted an $o(\log n)$-round algorithm.

We can, however, formally rule out natural approaches for solving this problem. For example, consider the approach of merging edges to obtain paths, retaining only the endpoints of the path (which is memory-efficient), and iterating to merge adjacent paths. For merging, note that paths sharing endpoints must be on the same machine. However, it seems hard to merge a super constant number of paths into a single path during a single round. Indeed, this idea can be formalized to show that it is hard to decrease the total number of paths stored across machines by more than a constant factor, which implies a lower bound of $\Omega(\log n)$ rounds for algorithms using this approach.

Generally, it is believed that the connectivity problem admits no $o(\log n)$-round algorithm. However, proving such a statement would resolve a long standing open problem in computational complexity. In this section, we will first discuss an unconditional lower bound showing that graph connectivity (and many other graph problems) need $\Omega(\log_M n)$ rounds, and improving this lower bound would imply a breakthrough result in circuit complexity theory, i.e., $\mathrm{NC}^1 \neq \mathrm{P}$. We then discuss conditional lower bounds, showing a reduction from maximum matching and maximal independent set problems to the graph connectivity, and showing that they require $\Omega(\log \log n)$ rounds.

8.2 Unconditional Lower Bounds

We first discuss why traditional computational complexity tools are hard to use to obtain unconditional lower bounds. Consider communication complexity, a standard lower bound tool, which quantifies the amount of communication needed between different parties owning different pieces of the input to jointly compute a function. Given that the difficulty of the graph connectivity comes from each machine seeing only part of the input, communication complexity seems appealing. However, in an MPC setting, the total amount of communication is allowed to exceed the input size. Unfortunately, communication complexity becomes meaningless here since the entire input can be communicated.

Circuit complexity offers a different set of tools for showing lower bounds. A (Boolean) circuit takes n binary inputs, $(x_1, \ldots, x_n) \in \{0, 1\}^n$, processes them via interconnected gates, such as 'and', 'or', 'not', and produces a binary output. The depth of a gate is the length of the longest path from any input to the gate and the depth of the circuit is the maximum depth of all gates. If every gate can take at most f inputs, then the circuit's fan-in is f. In the MPC model, while each machine's computation in round r can be viewed a "super" gate of depth r in a circuit, there are fundamental differences between MPC and circuits. First, a circuit is a static object while in MPC, depending on the local computation result, a machine can send data to different machines in the shuffle phase. Second, in MPC, a machine can choose to (be silent and) send out no data after the local computation, while each gate in a circuit always has an output.

The two differences are closely related to each other. Obviously, dynamic communication topology can give more power to MPC and it becomes more effective with silence being allowed since it can reduce unnecessary communication. Further, silence itself can carry extra information—for example, receiving no data from machine i could imply that the machine does not have the data that is desired. In fact, a Boolean circuit computation can be simulated in MPC, essentially meaning that MPC is more powerful than circuit computation.

Lemma 8.1. Every Boolean circuit of depth d and fan-in 2 can be simulated in $O(d/\log M)$ MPC rounds.

Proof. Each gate of depth $\log_2 M$ depends on at most M inputs as each gate has fan-in 2. It is easy to see that all gates of up to depth $\log_2 M$ can be simulated in one round of MPC. Repeating this completes the proof. □

The contrapositive of this immediately yields the following.

Corollary 8.2. If a problem P in P requires $\omega(\log n/\log M = \log_M n)$ MPC rounds, then the problem admits no Boolean circuit of depth $O(\log n)$ and fan-in 2, i.e., $P \notin \mathrm{NC}^1$.

It is a long-standing open question if $\mathrm{P} \neq \mathrm{NC}^1$. Thus, we cannot expect a better than $\Omega(\log_M n)$ lower bound without a breakthrough in circuit complexity.

8.2.1 Overview of $\Omega(\log_M n)$ Lower Bound

In the remainder of this section, we sketch an $\Omega(\log_M n)$ lower bound. The high-level strategy will consist of the following steps.

1. Defining an M-shuffle computation, which is more powerful than the MPC model; hence any lower bound for the M-shuffle model would imply the same lower bound for MPC. Roughly speaking, the model allows unlimited number of machines for computation and exponential time local computation.

2. Showing any R-round computation of the M-shuffle model can be expressed as a Boolean function that is a multi-linear polynomial of degree at most M^R.

3. Showing most graph properties, including connectivity, require high-degree polynomials.

8.2.2 M-Shuffle Model

Defining a model that is simpler yet more powerful than MPC will help obtain the desired lower bounds. To highlight the main ideas, we only present the important features of the M-shuffle model. The key insight to the modeling is to explicitly account for a machine *not* communicating with another machine through the use of a special symbol, $\perp$.

- There is no limit on the number of machines available. Thus, we can assume that each machine u is used only in a specific round $r(u) \in \{1, \ldots, R\}$, where R is the number of rounds.

- Each machine has an ordered set $\{1, \ldots, M\}$ of input ports.

- In each round, each machine u sends a message $\alpha_{uv}(\cdot) \in \{0, 1, \perp\}^M$ to each machine v with $r(v) > r(u)$.
- Machine v receives 0, 1, or $\perp$ on its ith input port from every machine u with $r(u) < r(v)$. If the machine receives more than one non $\perp$ symbol on each port, the computation is declared invalid. Otherwise, machine v's input, denoted as $g(v)$, is naturally defined: $g(v)_i$ is 1 (0, resp.) if and only if its ith port receives a 1 (0, resp.); otherwise $\perp$.

We assume for ease of analysis that each machine's input ports are ordered. We leave it as an exercise to show that the assumption can be removed and how the model can extend to capture randomization.

8.2.3 Representing M-Shuffle Computation by Polynomials

Now we show that any R-round computation of the M-shuffle model can be expressed as a polynomial of degree at most M^R. We sketch the main idea here. Let $\mathbb{I}(\cdot)$ be the binary indicator function. We would like to show that for any machine v and any specific input z' to the machine, $\mathbb{I}(g(v) = z')$ can be expressed as a polynomial of degree at most $M^{r(v)}$.

To illustrate the idea, we only show that $\mathbb{I}(g(v) = z')$ is a polynomial of degree at most M^2 for a fixed machine v from round 2. Repeating this argument will give the desired claim: a machine from round 3 will take messages from a set of machines from round 1 or 2 and take their aggregation as its input. Let $f_{uit}(x)$ be a Boolean function that has value 1 if and only if machine u sends a symbol $t \in \{0, 1, \perp\}$ to machine v on port i when the input is x. Machine u is from round 1 and has memory M bits, so can depend on at most M input variables. Using $y_1 \wedge y_2 = y_1 y_2$, $y_1 \vee y_2 = (1 - y_1)(1 - y_2)$, $\neg y_1 = 1 - y_1$, $y_1 = y_1^2$ for any $y_1, y_2 \in \{0, 1\}$, it is easy to see that f_{uit} can be expressed as a polynomial of degree M. Then, we have

$$\mathbb{I}(g(v)_i = 0) = \sum_{u \in U} f_{ui0}(x), \quad \mathbb{I}(g(v)_i = 1) = \sum_{u \in U} f_{ui1}(x),$$
$$\mathbb{I}(g(v)_i = \perp) = 1 - \mathbb{I}(g(v)_i = 0) - \mathbb{I}(g(v)_i = 1),$$

where we used the fact that v receives at most one non-$\perp$ symbol on port i. Thus, we have,

$$\mathbb{I}(g(v) = z') = \prod_{i \in [M]} \mathbb{I}(g(v)_i = z'_i),$$

which proves that $\mathbb{I}(g(v) = z')$ can be represented as a polynomial of degree M^2, as desired.

8.2.4 Graph Connectivity is a High Degree Polynomial

It is well-known that any Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be represented as a multilinear polynomial $\sum_{S \subseteq [n]} h(S) x^S$ where $h(S) \in \{0,1\}$ and $x^S := \prod_{i \in S} x_i$, using the fact that $x_i^2 = x_i$. For a given f, this multilinear polynomial is unique and its degree is called the *degree* of f.

Consider the following simple graph problem. There are $n+1$ vertices, indexed $0, 1, 2, \ldots, n$. Edges can be present only between vertices $i-1$ and i for $i \in [n]$ and x_i indicates the presence or absence of the edge $(i-1, i)$. Then, the Boolean formula $\prod_{i=1}^{n} x_i$ of degree n variables has value 1 if and only if the graph is connected. As shown before, an R-round M-shuffle can compute a polynomial of degree at most M^R, and therefore, it immediately follows that even this special case of graph connectivity requires $\Omega(\log_M n)$ MPC rounds.

In general, graph connectivity is one example of a monotone graph property. Recall that for undirected graphs, a graph property is independent of the vertex labeling, is monotone if it becomes false when a new edge is added to the graph, and is non-trivial unless the property holds for all graphs or does not hold for any graph. It is known that any non-trivial monotone graph property has $\Omega(n^2)$ decision-tree complexity, and any Boolean function f of degree d has decision-tree complexity at most $2d^4$. Together these imply that computation of any monotone graph property requires $\Omega(\log_M n)$ rounds in the M-shuffle model.

8.3 Conditional Lower Bounds

Due to the barriers in obtaining a logarithmic lower bound for Problem 8.1.1, there have been efforts to prove super-constant lower bounds for other graph problems, assuming a logarithmic lower bound for connectivity. We highlight some of these recent results. The first is a conditional lower bound for s-t-connectivity in terms of the diameter.

Theorem 8.3. If there is no $o(\log n)$-round MPC algorithm for Problem 8.1.1, then there is no $o(\log D)$-round MPC algorithm for solving s-t-connectivity problem on graphs of diameter D.

Conditional lower bounds for many graph problems can also be obtained.

Theorem 8.4. If there is no $o(\log n)$-round MPC algorithm for Problem 8.1.1, then there is no $o(\log \log n)$-round MPC algorithm that computes an $O(1)$-approximate maximum matching, an $O(1)$-approximate vertex cover, or a maximal independent set.

These conditional lower bounds are obtained using a novel connection between the graph connectivity problem and the LOCAL model of computation, described next. In the LOCAL model, each vertex of a given graph only knows its own identity, the identity of its neighbors, and the total number of vertices; in particular, it does not know the whole graph. In each round, each vertex performs some computation based on its knowledge and then sends messages to its neighbors. At the end of these distributed steps of computation, each vertex needs to know its part of the output. For example, in the maximum matching problem, each vertex needs to know the vertex to which it is matched (if at all). In the maximal independent set problem, each vertex needs to know if it indeed belongs to the independent set.

The connection can be informally described as follows.

Theorem 8.5. Suppose a graph problem has a D-round lower bound in the LOCAL model, where $D \leq \log^{\gamma} n$ for some constant $\gamma < 1$. Then, there is no $o(\log D)$-round MPC algorithm for the problem unless there is an $o(\log n)$-round algorithm for Problem 8.1.1.

Theorem 8.4 then immediately follows from applying the known $\Omega(\sqrt{\log n / \log \log n})$-round lower bound in the LOCAL model for the graph problems in the statement.

The proof of Theorem 8.5 is quite involved; we only provide a high-level description of the main ideas. Suppose a graph problem has a D-round lower bound in the LOCAL model. Then, the idea is to show that if there is an MPC algorithm A solving this graph problem in $o(\log D)$ rounds, then this algorithm must see certain "far away" information (i.e., information that lies more than D hops away) to produce the correct result. Specifically, one can construct two graphs G and G' with specialized vertices v and v' such that G and G' are isomorphic within D hops from v and v' respectively, but the output at v and v' are different when running A. Using this, the idea is to construct an $o(\log D)$-round MPC algorithm for solving the D-diameter s-t-connectivity problem by using the $o(\log D)$-round algorithm A as an oracle, which will contradict Theorem 8.3.

8.4 Section Notes

The lower bounds and the formalization of the M-Shuffle was introduced by Roughgarden *et al.* (2016), the conditional lower bounds are due to Ghaffari *et al.* (2019a). Finally, Charikar *et al.* (2020) showed an unconditional lower bound of $\Omega(\log_M n)$ for the 1-vs-2-cycle problem by using a connection between query complexities and Boolean formula degrees.

Many more results are known for restricted settings. Im and Moseley (2019) showed that a certain class of algorithms require $\Omega(\log n)$ rounds to solve the 1-vs-2-cycle problem. A related result was shown earlier by Beame *et al.* (2017) in the context of relational data sets. A lower bound under similar algorithmic limitations was given for a certain game by Jacob *et al.* (2014).

For specific problems, Pietracaprina *et al.* (2012) gave a lower bound for matrix multiplication assuming that all elementary products must be computed. Bilardi *et al.* (2012) gave tight lower bounds on the fan-in for computing the Fast Fourier Transform in a restricted BSP model.

9

Conclusions

The MPC model has emerged as a valuable formalization of modern parallel computing, encompassing many different frameworks and implementations, such as MapReduce, Hadoop, and Spark, among others. The sublinear memory constraint, coupled with the goal of minimizing the total number of rounds forces the algorithm designer to parallelize the computation not too coarsely (respecting the memory constraints), and not too finely (keeping to a small number of rounds). This sweet-spot has led to new algorithms that we have covered in this monograph.

We have presented a lot of different techniques that are used in design of MPC algorithms, but the coverage is not comprehensive by design. We have chosen approaches that unify multiple results and identify the key algorithmic and analytical insights. Our focus was on exploring and understanding these techniques; for many problems we deliberately did not present state-of-the-art results.

The field of MPC algorithms continues to be actively pursued by the research community and the demand for new and more efficient MPC algorithms continues to rise among practitioners. We hope this survey will be useful to both these communities and will act as a conduit for new problems and new algorithms.

References

Afrati, F. N., A. D. Sarma, S. Salihoglu, and J. D. Ullman. (2012). “Upper and Lower Bounds on the Cost of a Map-Reduce Computation”. arXiv: 1206.4377.

Andoni, A., A. Nikolov, K. Onak, and G. Yaroslavtsev. (2014). “Parallel Algorithms for Geometric Graph Problems”. In: *STOC*. 574–583.

Andoni, A., Z. Song, C. Stein, Z. Wang, and P. Zhong. (2018). “Parallel Graph Connectivity in Log Diameter Rounds”. In: *FOCS*. 674–685.

Assadi, S., M. Bateni, A. Bernstein, V. S. Mirrokni, and C. Stein. (2019a). “Coresets Meet EDCS: Algorithms for Matching and Vertex Cover on Massive Graphs”. In: *SODA*. 1616–1635.

Assadi, S., M. Bateni, and V. S. Mirrokni. (2019b). “Distributed Weighted Matching via Randomized Composable Coresets”. In: *ICML*. 333–343.

Bahmani, B., R. Kumar, and S. Vassilvitskii. (2012a). “Densest Subgraph in Streaming and MapReduce”. *PVLDB*. 5(5): 454–465.

Bahmani, B., B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. (2012b). “Scalable K-Means++”. *PVLDB*. 5(7): 622–633.

Bateni, M. H., S. Behnezhad, M. Derakhshan, M. T. Hajiaghayi, R. Kiveris, S. Lattanzi, and V. Mirrokni. (2017). “Affinity clustering: Hierarchical clustering at scale”. In: *NIPS*. 6867–6877.

Beame, P., P. Koutris, and D. Suciu. (2017). “Communication steps for parallel query processing”. *JACM*. 64(6): 40:1–40:58.

Behnezhad, S., L. Dhulipala, H. Esfandiari, J. Lacki, and V. S. Mirrokni. (2019a). “Near-Optimal Massively Parallel Graph Connectivity”. In: *FOCS*. 1615–1636.

Behnezhad, S., M. Hajiaghayi, and D. G. Harris. (2019b). “Exponentially Faster Massively Parallel Maximal Matching”. In: *FOCS*. 1637–1649.

Bhaskara, A. and M. Wijewardena. (2018). “Distributed Clustering via LSH Based Data Partitioning”. In: *ICML*. 569–578.

Bilardi, G., M. Scquizzato, and F. Silvestri. (2012). “A lower bound technique for communication on BSP with application to the FFT”. In: *ECPP*. 676–687.

Broder, A. Z., L. G. Pueyo, V. Josifovski, S. Vassilvitskii, and S. Venkatesan. (2014). “Scalable K-Means by ranked retrieval”. In: *WSDM*. 233–242.

Ceccarello, M., A. Pietracaprina, and G. Pucci. (2019). “Solving k-center Clustering (with Outliers) in MapReduce and Streaming, almost as Accurately as Sequentially”. *Proc. VLDB Endow.* 12(7): 766–778.

Chang, Y., M. Fischer, M. Ghaffari, J. Uitto, and Y. Zheng. (2019). “The Complexity of $(\Delta{+}1)$ Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation”. In: *PODC*. 471–480.

Charikar, M., W. Ma, and L.-Y. Tan. (2020). “New lower bounds for Massively Parallel Computation from query complexity”. In: *SPAA*. 141–151.

Chierichetti, F., R. Kumar, and A. Tomkins. (2010). “Max-cover in map-reduce”. In: *WWW*. 231–240.

Cohen-Addad, V., S. Lattanzi, S. Mitrovic, A. Norouzi-Fard, N. Parotsidis, and J. Tarnawski. (2021a). “Correlation Clustering in Constant Many Parallel Rounds”. In: *ICML*. 2069–2078.

Cohen-Addad, V., S. Lattanzi, A. Norouzi-Fard, C. Sohler, and O. Svensson. (2021b). “Parallel and Efficient Hierarchical k-Median Clustering”. In: *NeurIPS*.

Coy, S. and A. Czumaj. (2022). “Deterministic massively parallel connectivity”. In: *STOC*. 162–175.

Czumaj, A., J. Lacki, A. Madry, S. Mitrovic, K. Onak, and P. Sankowski. (2018). “Round compression for parallel matching algorithms”. In: *STOC*. 471–484.

Dean, J. and S. Ghemawat. (2008). “MapReduce: Simplified Data Processing on Large Clusters”. *CACM*. 51: 107–113.

Ene, A., S. Im, and B. Moseley. (2011). “Fast clustering using MapReduce”. In: *KDD*. 681–689.

Ene, A. and H. L. Nguyen. (2015). “Random Coordinate Descent Methods for Minimizing Decomposable Submodular Functions”. In: *ICML*. 787–795.

Esfandiari, H., S. Lattanzi, and V. Mirrokni. (2018). “Parallel and Streaming Algorithms for K-Core Decomposition”. In: *ICML*. 1396–1405.

Feldman, J., S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. (2010). “On distributing symmetric streaming computations”. *TALG*. 6(4): 66:1–66:19.

Frigo, M., C. E. Leiserson, H. Prokop, and S. Ramachandran. (2012). “Cache-Oblivious Algorithms”. *TALG*. 8(1): 4.

Ghaffari, M., T. Gouleakis, C. Konrad, S. Mitrovic, and R. Rubinfeld. (2018). “Improved Massively Parallel Computation Algorithms for MIS, Matching, and Vertex Cover”. In: *PODC*. 129–138.

Ghaffari, M., F. Kuhn, and J. Uitto. (2019a). “Conditional Hardness Results for Massively Parallel Computation from Distributed Lower Bounds”. In: *FOCS*. *FOCS ’19*.

Ghaffari, M., S. Lattanzi, and S. Mitrovic. (2019b). “Improved Parallel Algorithms for Density-Based Network Clustering”. In: *ICML*. 2201–2210.

Ghaffari, M. and J. Uitto. (2019). “Sparsifying Distributed Algorithms with Ramifications in Massively Parallel Computation and Centralized Local Computation”. In: *SODA*. 1636–1653.

Goel, A. and K. Munagala. (2012). “Complexity Measures for Map-Reduce, and Comparison to Parallel Computing”. arXiv: 1211.6526.

Gonzalez, T. F. (1985). “Clustering to minimize the maximum intercluster distance”. *TCS*. 38: 293–306.

Goodrich, M. T. (2010). “Simulating Parallel Algorithms in the MapReduce Framework with Applications to Parallel Computational Geometry”. arXiv: 1004.4708.

Goodrich, M. T., N. Sitchinava, and Q. Zhang. (2011). “Sorting, Searching, and Simulation in the Mapreduce Framework”. In: *ISAAC*. 374–383.

Hegeman, J. W. and S. V. Pemmaraju. (2015). “Lessons from the congested clique applied to MapReduce”. *TCS*. 608: 268–281.

Im, S. and B. Moseley. (2015). “Brief Announcement: Fast and Better Distributed MapReduce Algorithms for k-Center Clustering”. In: *SPAA*. 65–67.

Im, S. and B. Moseley. (2019). “A Conditional Lower Bound on Graph Connectivity in MapReduce”. arXiv: 1904.08954.

Im, S., B. Moseley, and X. Sun. (2017). “Efficient massively parallel methods for dynamic programming”. In: *STOC*. 798–811.

Indyk, P., S. Mahabadi, M. Mahdian, and V. S. Mirrokni. (2014). “Composable core-sets for diversity and coverage maximization”. In: *PODS*. 100–108.

Jacob, R., T. Lieber, and N. Sitchinava. (2014). “On the complexity of list ranking in the parallel external memory model”. In: *MFCS*. 384–395.

Karloff, H., S. Suri, and S. Vassilvitskii. (2010). “A Model of Computation for MapReduce”. In: *SODA*. 938–948.

Koutris, P., S. Salihoglu, and D. Suciu. (2018). “Algorithmic Aspects of Parallel Data Processing”. *Foundations and Trends in Databases*. 8(4): 239–370.

Kumar, R., B. Moseley, S. Vassilvitskii, and A. Vattani. (2015). “Fast Greedy Algorithms in MapReduce and Streaming”. *TOPC*. 2(3): 14:1–14:22.

Lattanzi, S., T. Lavastida, K. Lu, and B. Moseley. (2019). “A framework for parallelizing hierarchical clustering methods”. In: *ECML/PKDD*. 73–89.

Lattanzi, S., B. Moseley, S. Suri, and S. Vassilvitskii. (2011). “Filtering: a method for solving graph problems in MapReduce”. In: *SPAA*. 85–94.

Lenzen, C. and R. Wattenhofer. (2010). “Brief announcement: exponential speed-up of local algorithms using non-local communication”. In: *PODC*. 295–296.

Linial, N. (1987). “Distributive graph algorithms global solutions from local data”. In: *FOCS*. 331–335.

Malkomes, G., M. J. Kusner, W. Chen, K. Q. Weinberger, and B. Moseley. (2015). “Fast Distributed k-Center Clustering with Outliers on Massive Data”. In: *NIPS*. 1063–1071.

McGregor, A. (2014). “Graph stream algorithms: a survey”. *SIGMOD Record*. 43(1): 9–20.

Mirzasoleiman, B., A. Karbasi, R. Sarkar, and A. Krause. (2013). “Distributed Submodular Maximization: Identifying Representative Elements in Massive Data”. In: *NIPS*. 2049–2057.

Mitzenmacher, M. and E. Upfal. (2005). *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press.

Nemhauser, G. L., L. A. Wolsey, and M. L. Fisher. (1978). “An Analysis of Approximations for Maximizing Submodular Set Functions–I”. *Math. Program.* 14(1): 265–294.

Park, H.-M., F. Silvestri, U. Kang, and R. Pagh. (2014). “MapReduce Triangle Enumeration With Guarantees”. In: *CIKM*. 1739–1748.

Pietracaprina, A., G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. (2012). “Space-round Tradeoffs for MapReduce Computations”. In: *ICS*. 235–244.

Ponte Barbosa, R. da, A. Ene, H. L. Nguyen, and J. Ward. (2015). “The Power of Randomization: Distributed Submodular Maximization on Massive Datasets”. In: *ICML*. Vol. 37. 1236–1244.

Ponte Barbosa, R. da, A. Ene, H. L. Nguyen, and J. Ward. (2016). “A New Framework for Distributed Submodular Maximization”. In: *FOCS*. 645–654.

Roughgarden, T., S. Vassilvitskii, and J. T. Wang. (2016). “Shuffles and Circuits (On Lower Bounds on Massively Parallel Computation)”. In: *SPAA ’16*.

Suri, S. and S. Vassilvitskii. (2011). “Counting triangles and the curse of the last reducer”. In: *WWW*. 607–614.

Valiant, L. G. (1990). “A bridging model for parallel computation”. *CACM*. 33(8): 103–111.

Williamson, D. P. and D. B. Shmoys. (2011). *The Design of Approximation Algorithms*. Cambridge University Press.

Zhao, Z., G. Wang, A. Butt, M. Khan, V. Kumar, and M. Marathe. (2012). "SAHAD: Subgraph Analysis in Massive Networks Using Hadoop". In: *IPDPS*. 390–401.