Efficient Parallel Approximation Algorithms

Kanat Tangwongsan

CMU-CS-11-128

August 2011

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch, Co-Chair
Anupam Gupta, Co-Chair
Avrim Blum
Gary L. Miller
Satish Rao, University of California, Berkeley

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2011 Kanat Tangwongsan

This work is sponsored in part by the National Science Foundation under grant numbers CCF-1018463, CCF- 1018188, and CCF-1016799, by an Alfred P. Sloan Fellowship, and by generous gifts from IBM, Intel Coporation, and Microsoft Coporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF, IBM, Intel, or Microsoft, or other funding parties.

To my parents, Chinda (Achariyakul) Tangwongsan and Supachai Tangwongsan,
and to my grandparents,
who have instilled in me curiosity and courage to discover things
and have always supported me in everything I have attempted.

Abstract

Over the past few decades, advances in approximation algorithms have enabled near-optimal solutions to many important, but computationally hard, problems to be found efficiently. But despite the resurgence of parallel computing in recent years, only a small number of these algorithms have been considered from the standpoint of parallel computation. Furthermore, among those for which parallel algorithms do exist, the algorithms—mostly developed in the late 80s and early 90s—follow a design principle that puts heavy emphasis on achieving polylogarithmic depth, with little regard to work, generally resulting in algorithms that perform substantially more work than their sequential counterparts. As a result, on a modest number of processors, these highly parallel—but "heavy"—algorithms are unlikely to perform competitively with the state-of-the-art sequential algorithms. This motivates the question: *How can one design a parallel approximation algorithm that obtains non-trivial speedups over its sequential counterpart, even on a modest number of processors?*

In this thesis, we explore a set of key algorithmic techniques that facilitate the design, analysis, and implementation of a wide variety of efficient parallel approximation algorithms. This includes:

— *Maximal nearly independent set.* A natural generalization of maximal independent set (MIS) solvable in linear work, which leads to linear-work RNC $((1+\varepsilon)\ln n)$-approximation set cover, $(1-\frac{1}{e}-\varepsilon)$-approximation (prefix optimal) max cover, and $(4+\varepsilon)$-approximation min-sum set cover—and a work-efficient RNC $(1.861+\varepsilon)$-approximation for facility location.

— *Low-diameter decomposition, low-stretch spanning trees, and subgraphs.* This allows us to develop a near-linear work, $O(m^{1/3})$-depth algorithm for solving a symmetric diagonally dominant (SDD) linear system $Ax = b$, with m nonzeros. The solver leads fast parallel algorithms for max flow, min-cost flow, (spectral) sparsifier, etc.

— *Probabilistic tree embeddings.* An RNC $O(n^2 \log n)$-work algorithm for probabilistic tree embeddings with expected stretch $O(\log n)$, independent of the aspect ratio of the input metric. This is a parallel version of Fakcharoenphol et al.'s algorithm, providing a building block for algorithms for k-median and buy-at-bulk network.

— *Hierarchical diagonal blocking.* A sparse matrix representation that exploits the small separators property found in many real-world matrices. We also develop a low-depth parallel algorithm for the representation, which achieves substantial speedups over existing SpMV code.

Acknowledgments

I am deeply grateful to my advisors—Guy Blelloch and Anupam Gupta—for making 5 years of graduate school one of the best periods of my life. I really could not ask for a better or more caring advisor. Over the years, both Guy and Anupam have given me the freedom to spend my time on various (independent) projects while always giving me wonderful advice and making sure that I don't lose sight of the big picture. I thank them for putting up with my random and often unfruitful ideas, all the while helping me distill legit ideas from a sea of crap, and for believing that something good will come out eventually. Many times during the course of the PhD I went into a meeting with them frustrated and losing hope on research work; thanks for giving me the excitement to persevere despite the lack of concrete progress.

I'm also indebted to my committee members, Avrim Blum, Gary Miller, and Satish Rao, for seeing this thesis through, for letting me steal many hours of their much-demanded time, and for invaluable questions and suggestions that lead to various results in this thesis.

The results in this thesis are joint with Guy Blelloch, Anupam Gupta, Yiannis Koutis, Gary Miller, Richard Peng, and Harsha Simhadri. Thank you for your brilliant ideas and making work fun to the extent possible. In particular, thank you, Richard—late night and weekend head banging parties, feeding frenzies, the tale of 12 days of Christmas, cow- and munchkins-related humor, etc., are fun times and fond memories.

Beside work included in this thesis, I have had the pleasure of working with a number of great minds on various projects (in no particular order): Umut Acar, Dave Andersen, Daniel Golovin, Amit Kumar, Bindu Pucha, Michael Kaminsky, Ruy Ley-Wild, Mathias Blume, Bob Harper, Duru Türkoğlu, and Jorge Vittes. Umut, thank you for getting me started on research, for showing me how rewarding and enjoyable it can be to work on problems you are passionate about, for giving a number of cool problems to work on, and for convincing me to stay at Carnegie Mellon for graduate school. I owe thanks to Dave Andersen, who has directly and indirectly influenced my work over the years—thank you for helping me in a fun system project in every way imaginable (millions of thanks to Bindu as well); for teaching me about systems research and paper writing; and most importantly, for helping me realize that the theory-systems blend is a load of fun. I'll consider myself lucky if I can write half as well as Dave Andersen does. Moreover, the Computer Science department at Carnegie Mellon has been an invaluable resource for me; I couldn't imagine a more stimulating and collaborative environment to conduct research in.

To Mom and Dad: I would not be where I am today without your unwavering support, encouragement, and love—and blessing from family members. Thank you for always being there when I need it, for understanding why a PhD takes N (generally more than 4) years to finish (and not rushing it), and for believing in me—despite my cat insanity—in this whole endeavor. Thanks for being the perfect example of hardworking spirit, and instilling in me curiosity and courage to learn about and discover new things.

To friends and colleagues: This thesis would not be what it is without you guys and your constant support, moral and otherwise. You have my eternal gratitude, though it isn't possible to make a complete list of everyone (as it will be drastically incomplete, however hard I try). But I feel the urge to single out (and embarrass) some of you in no thoughtful order. Lek Viriyasitavat, you have made these many years in Pittsburgh feel much like home; thank you so much for tolerating my nonsense, for feeding me many delicious meals, for offering a kind shoulder to lean on, and in fact for just being around. I cannot thank Kwang Pongnumkul enough for being super awesome at everything, for offering me practical advice and support and lending me ears countless times, for smiles and laughter from fun conversations, for keeping my company (remotely) when I had to work late at night, and for dragging me out of hell when I fell in one. I'm very fortunate to have someone close to go through the PhD experience with. I realize that during difficult times, you probably believe in me more so than I believe in myself; once again, thank you. Ning Kiatpaiboon, thanks for saving (the last bit of) my sanity :) through the final years of graduate school and for various soul-searching conversations that make me understand many things better. Yinmeng Zhang, thank you for being the best kind of bad influence, for inviting me over for great food and distractions, for pointing me to bits and pieces of useful math, and for your patience when I whine about things.

Furthermore, my officemates over the year—Leonid Kontorovich, Haowen Chan, B. Aditya Prakash, Kate Taralova, and Stephanie Rosenthal—have made my hours at work enjoyable. Friends both in Pittsburgh and elsewhere make this journey a really special one. Thank you, everyone, especially Fern Kundhikanjana, Kornchawal Chaipah, Aaron Roth, Arbtip Dheeravongkit, Gwendolyn Stockman, Roy Liu, Vijay Vasudevan, Noey Munprom, Naju Mancheril, March Mahatham, Mike Dinitz, Don Sheehy, Yi Wu, Amar Phanishayee, Ig Chuengsatiansup, Chatklaw Jareanpon, Pai Sriprachya-anunt, Orathai Sukwong, Akkarit Sangpetch, Mock Suwannatat, Him Dhangwatnotai, Mihir Kedia, … (I apologize for not having room to mention your name here.)

Finally, I thank 21st Coffee and Tazza D'oro, and Crazy Mocha and Coffee Tree Roasters (for the first few years), where much of this work got done.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel machines have become ubiquitous in the past several years, so much so that it is nearly impossible to buy sequential machines today. Even the latest generation of cell phones have multiple general-purpose cores, along with additional special-purpose parallel hardware. Parallel machines with tens of cores are already common—and machines with hundreds, if not thousands, of cores are expected in the mainstream in a few years, with the number predicted to soar several orders of magnitude more in a decade. This trend, driven in part by physical and economical constraints of processor chips' production and energy consumption, has had profound implications to how algorithms must be designed and implemented to fully leverage the increasing amount of parallelism.

During a similar time frame, the computing industry has witnessed an unprecedented growth in data size, as well as in problem complexity, which demands more computational power than ever. This trend has drawn the attention of several communities to approximation algorithms, whose advances over the last few decades have allowed the computation of provably near-optimal solution (e.g., within less than 0.1% of the optimal solution) in a fraction of the time required to compute an optimal solution. Despite significant progress made on this front, these algorithms generally have not been considered from the standpoint of parallel algorithms. For the past decades, the community has focused mainly on improving the approximation guarantees and running time of sequential algorithms.

This thesis work lies at the interface of parallel algorithms and approximation algorithms, initiating a principled study of the design, analysis, and implementation of *efficient parallel approximation algorithms*—approximation algorithms that can *efficiently* take advantage of parallelism. By contrast, early work on parallel approximation algorithms is less concerned with efficiency aspects. Back in the 80s and the early 90s, the community advocated a design principle that puts heavy emphasis on achieving polylogarithmic depth[1] with little regard to work. This results in a number of beautiful algorithms that have an impressive amount of parallelism but perform substantially more work than their sequential counterparts. Unfortunately, on a modest number of processors, these highly parallel—but "heavy"—algorithms are unlikely to perform competitively with the state-of-the-art sequential algorithms.

The present work takes a fresh look at this fascinating decades-old subject with a particular focus on efficiency. Specifically, we are most concerned with the question:

> *How can one design a parallel approximation algorithm that obtains non-trivial speedups over its sequential counterpart, even on a modest number of processors?*

To answer this question, we set out to identify and develop a set of key techniques to facilitate the development of such algorithms. We believe the right combination of parallelism and approximation algorithms is key to getting scalable performance and is our hope in keeping pace with the growing demand. We adopt a now-common design principle that strives for low depth—and more importantly work efficiency (i.e., the algorithm cannot perform significantly more work than its sequential counterpart does). Following this guideline, we ask:

(1) What key techniques in approximation algorithms are efficiently parallelizable?

(2) How can they be applied to parallelize existing approximation algorithms?

(3) How can algorithmic techniques help overcome challenges arising in the implementation of these algorithms on a modern machine architecture?

Before we give an overview of the results in this thesis, we present a summary of prior work on parallel approximation algorithms to put this work in perspective.

1.1 Early Work on Parallel Approximation Algorithms

Throughout the late 1980s and early 1990s, research in the algorithms community focused most of the efforts on developing parallel algorithms, some of which are approximation algorithms, an area which started to gain popularity around the time interest in parallel comput-

[1]By depth, we mean the longest chain of dependencies and by work, the total operation count. We formally define these notions in Chapter 2)

ing research started to fade. It should be noted, however, that back then, the primary goal was to obtain NC and RNC algorithms (i.e., algorithms that run in polylogarithmic depth) even at the expense of much worse work bounds. This trend was in large part motivated and justified by work in complexity theory that popularizes the distinction between (R)NC and P, where polylogarithmic depth is often thought of as fast running time and work is of secondary concern, as long as it is kept polynomial.

During this period of time, several RNC and NC algorithms have been proposed. There are RNC and NC parallel approximation algorithms for set cover [BRS94, RV98], vertex cover [KVY94, KY09], special cases of linear programs (e.g., positive LPs and cover-packing LPs) [LN93, Sri01, You01], and k-center [WC90]. These algorithms are typically based on parallelizing their sequential counterparts, which usually contain an inherently sequential component (e.g., a greedy step which requires picking and processing the minimum-cost element before proceeding to the next). A common idea in these parallel algorithms is that instead of picking only the most cost-effective element, they make room for parallelism by allowing a small slack (e.g., a $(1+\varepsilon)$ factor) in what can be selected. This idea often results in a slightly worse approximation factor than the sequential version. For instance, the parallel set-cover algorithm of Rajagopalan and Vazirani is a $(2(1+\varepsilon)\ln n)$-approximation, compared to a $\ln n$-approximation produced by the standard greedy set cover. Likewise, the parallel vertex-cover algorithm of Khuller et al. is a $2/(1-\varepsilon)$-approximation as opposed to the optimal 2-approximation given by various known sequential algorithms. Only recently has the approximation factor for vertex cover been improved to 2 in the parallel case [KY09].

1.2 Thesis Overview

As a starting point in understanding how techniques in approximation algorithms can be parallelized efficiently, we revisit basic problems in approximation algorithms with well-understood sequential algorithms. Our starting point is a small, but diverse, set of results in approximation algorithms for problems such as facility location, set cover, and max k-cover, with a primary goal of developing techniques for devising their efficient parallel counterparts.

Facility Location. Facility location is an ideal class of problems to begin this study. Not only are these problems important because of their practical value, but they appeal to study because of their special stature as "testbeds" for techniques in approximation algorithms. As such, deriving parallel algorithms for these problems is an invaluable step in understanding how to parallelize other algorithms. We present several algorithms based on greedy, primal-dual, local-search, and LP-rounding techniques. These results are summarized in Section 1.3.

Lessons from this study were instrumental in shaping the formulation of the maximal nearly independent set problem, which forms the basis for improved algorithms for set cover, max k-cover, min-sum set cover, and even greedy facility location itself.

Maximum Cut. In addition to facility location, we consider other standard problems in approximation algorithms, in hope that studying them will shed light on how to parallelize similar algorithms. For this, we look at the maximum cut (MaxCut) problem: while the simple randomized $\frac{1}{2}$-approximation is trivial to parallelize, it is non-trivial to obtain an RNC factor-$(\alpha_{GW} - \varepsilon)$ algorithm, where α_{GW} is approximation ratio obtained by the state-of-the-art Goemans and Willamson's algorithm [GW95]. The standard GW implementation requires solving a semidefinite program (SDP), which in general is P-hard. We describe in Section 1.4 an RNC near-linear work algorithm for MaxCut that achieves essentially the same guarantees as the GW algorithm.

Advances in approximation algorithms often result from ingenious applications of time-tested techniques in conjunction with insightful problem-specific ideas. Refined over time, these versatile techniques are applied to new problems time and again. For instance, techniques such as greedy and randomized rounding are used not only in set cover but also in facility location; and metric-embedding techniques are applied to k-median and buy-at-bulk network design, to name a few. This leads to the question: it possible to parallelize some of these techniques?

Maximal Nearly Independent Set. A generalization of Maximal Independent Set (MIS), this formulation is inspired by greedy parallel algorithms for set cover and facility location, and the following simple observation: while the greedy process is often inherently sequential, one can create opportunities for parallelism by choosing not only the best option available but also every option roughly as good as the best option, within a threshold. This idea turns out to be too aggressive: the chosen options may interfere with one another, leading to low-quality solutions. To rectify this, we formulate and study a combinatorial problem—called Maximal Nearly Independent Set (MaNIS)—to capture the situation. We present a linear-work RNC algorithm for the problem, leading to linear-work RNC algorithms for set cover, prefix-optimal max cover, and min-sum set cover with essentially optimal approximation guarantees, and a work-efficient RNC $(1.861 + \varepsilon)$-approximation for facility location.

Embeddings of Distances. Another family of key technique in approximation algorithms is embeddings: mapping a problem instance into an "easier" space, where it is solved and "lifted" back to the original space. Embedding techniques underlie the design of several of the best known approximation algorithms. Of particular interest to this work are variants of the following problems: embedding metric spaces into distributions over a simpler metric space (e.g. trees and ultrasparse subgraphs). We present three results related to this. In Section 1.6 (and in details in Chapter 6), we discuss the first two results—how to embed the distance metric of an arbitrary (sparse) graph into a spanning subtree or an ultrasparse spanning subgraph. The latter is motivated by the problem of solving symmetric diagonally dominant (SDD) linear systems, which we also describe in Chapter 6.

The other embedding result is concerned with probabilistically embedding an arbitrary finite metric space into a distribution of dominating trees (a parallel version of the FRT embedding). This result is summarized in Section 1.7 (detailed description in Chapter 7). Using the embedding result, we present algorithms for k-median and buy-at-bulk network design. We believe there are numerous other applications of these embedding techniques.

The complexity of modern parallel machines presents opportunities for algorithmic innovations to improve computing performance. As an example, it is commonly observed that as the number of cores increases, the per-core memory bandwidth does not scale proportionally, starving the individual cores in memory-intensive applications. This phenomenon turns out to be a major limiting factor in the scalability of sparse matrix-vector multiply (SpMV), an important subroutine in high-performance computing and other data-intensive applications.

Low-Bandwidth Computing by Exploiting Structure in the Data. A surprising number of real-world datasets have a certain structure that can be exploited in algorithms design. As a prime example, real-world graphs, including the US road network, the Internet graph, and finite-element meshes, have "small separators[2]," a useful property that has led to the development of data compression techniques that require only a linear of number of bits and enhance locality [BBK04, BBK03]. Inspired by previous work in this area, we study how to take advantage of the small-separator structure to lower the bandwidth requirement. As a step in this direction, we propose a representation—called Hierarchical Diagonal Blocking (HDB)—which can substantially enhance the performance of SpMV. This work is summarized in Section 1.8. Already, improving the performance of SpMV automatically boosts the performance of the numerous applications that use SpMV at the core.

Parallel and I/O Algorithms for Set Cover and Related Problems. Finally, building on the results on maximal nearly independent set, we design and analyze I/O efficient and parallel versions of algorithms for set cover, max k-cover, and min-sum set cover, with I/O cost no more expensive than that of sorting. We demonstrate the practicality of these algorithms by showing empirical evidence that our algorithms are substantially faster than existing implementations. This is discussed in Section 1.9

[2]We give a precise definition in Section 2.2

1.3 Algorithms for Facility Location Problems

Studied in Chapter 3, facility location is an important and well-studied class of problems in approximation algorithms, with far-reaching implications in many application areas. As we discussed earlier, these problems are important from both practical and theoretical perspectives. For these reasons, deriving parallel algorithms for facility location problems was our first step towards understanding how to parallelize common techniques in approximation algorithms. Previous work on facility location commonly relies on techniques such as linear-program (LP) rounding, local search, primal dual, and greedy, for which no general-purpose parallelization technique exists.

We show how to parallelize many facility-location algorithms. Our goal was to understand how different techniques in approximation algorithms can be parallelized. To this end, we consider the primal-dual algorithm of Jain and Vazirani [JV01b], the greedy algorithm of Jain et al. [JMM$^+$03], and the LP-rounding algorithm of Shmoys et al. [STA97]. Additionally, we study Hochbaum and Shmoys's algorithm [HS85] for k-center, and the natural local-search algorithms for k-median and k-means [AGK$^+$04, GT08]. Details appear in Chapter 3. We summarize the results of this chapter in the following:

Primal-Dual Algorithm. Building on the sequential primal-dual algorithm of Jain and Vazirani [JV01a], we obtain the following theorem.

Theorem 1.1 *Let $\varepsilon > 0$ be fixed. For sufficiently large m, there is a primal-dual* RNC *$O(m \log_{1+\varepsilon} m)$-work algorithm that yields a factor-$(3+\varepsilon)$ approximation for the metric facility-location problem.*

Greedy Algorithm. The greedy scheme underlies an exceptionally simple algorithm for facility location, due to Jain et al. [JMM$^+$03]. To describe the algorithm, we need a couple of definitions: a star (i, S) consists of facility i and a set of clients S, costing $\frac{1}{|S|}(f_i + \sum_{j \in S} d(j,i))$. The greedy algorithm of Jain et al. proceeds as follows:

> Until no client remains, pick the cheapest star (i, C'), open the facility i, set $f_i = 0$, remove all clients in C' from the instance, and repeat.

Parallelizing this algorithm gives the following bounds:

Theorem 1.2 *Let $0 < \varepsilon \leq 1$ be fixed. For sufficiently large input, there is a greedy-style* RNC *$O(m \log^2_{1+\varepsilon}(m))$-work algorithm that yields a factor-$(3.722+\varepsilon)$ approximation for the metric facility-location problem.*

Results on Other Facility-Location Problems. In addition to these algorithms, we are able to obtain the following results:

- given an optimal LP solution for the standard primal LP, there is an RNC rounding algorithm yielding a $(4+\varepsilon)$-approximation with $O(m \log m \log_{1+\varepsilon} m)$ work (based on the sequential algorithm of Shmoys et al. [STA97]).
- a 2-approximation for k-center based on the algorithm of Hochbaum and Shmoys [HS85].
- for $k \leq \text{polylog}(n)$, $(5+\varepsilon)$- and $(81+\varepsilon)$- approximation algorithms for k-median and k-means, resp., based on the natural local search algorithms [AGK$^+$04, GT08]

Note that for the k-center result, we did not sacrifice the solution's quality in the process.

Summary of Techniques. Common in these algorithms (except for the k-center algorithm) is an idea we call *geometric-scale bucketing.* Most of these algorithms proceed in multiple rounds, where each round can be viewed as identifying and processing the best available option. Geometric scaling creates opportunities for parallelism by attempting to process every option that is roughly as good as the best option (up to some multiplicative factors). Consequently, it typically causes some notion of utility to increase in powers of $(1+\varepsilon)$, resulting in a small number of rounds.

In most cases, however, this policy turns out to be too aggressive, and we need another idea to control the solution's quality. This is typically different for each problem. Specifically, for the greedy facility-location algorithm, we need an idea to manage the overlaps between the chosen stars. This process forms the starting point of the work studied in Chapter 5.

1.4 Algorithms for Maximum Cut

Chapter 4 presents a parallel algorithm for the maximum cut (MaxCut) problem. This basic graph optimization problem has laid the foundation for many of the now-common techniques in both approximation algorithms and the theory of hardness of approximation. Given a graph $G = (V, E)$, the goal of this problem is to find a bipartition of the graph so as to maximize the number of edges between the parts.

We design and analyze a parallel algorithm for MaxCut that essentially matches the approximation ratio $\alpha_{GW} \approx 0.878\cdots$ of the algorithm of Goemans and Williamson [GW95]. The algorithm runs in nearly linear work and has polylogarithmic depth. More specifically, we show the following theorem:

Theorem 1.3 *For a fixed constant $\varepsilon > 0$, there is a $(1-\varepsilon)\alpha_{GW}$-approximation algorithm for*

MaxCut on an unweighted graph with n nodes and m edges that runs in $O(m \log^7 n)$ work and $O(\log^5 n)$ depth.

The main technical ideas include a parallel transformation and sparsification technique (part of it was implicit in [Tre01]) that turns an arbitrary unweighted graph into a graph with slighter more nodes but without any high-degree node, and a parallel implementation of Arora and Kale's primal-dual framework for approximately solving SDPs [AK07]. We believe our parallel implementation of the AK framework will find other interesting applications, especially when the SDP has small width.

1.5 Maximal Nearly Independent Set and Applications

In Chapter 5, we initiate the study of maximal nearly-independent set, a generalization of the well-known maximal independent set (MIS) problem. As motivation, we consider parallel greedy approximation algorithms for set cover and related problems. Sequentially, the greedy method for set cover iteratively chooses a set that has optimal cost per element and removes the set as well as its elements. For n ground elements, this gives a polynomial algorithm with an optimal (assuming standard complexity assumptions) approximation ratio of $H_n = \sum_{k=1}^{n} \frac{1}{k}$. Berger, Rompel, and Shor [BRS94] show that the method can be parallelized by bucketing costs by factors of $(1+\varepsilon)$ and processing sets within a bucket in parallel. This requires a careful subselection within a bucket so that the sets selected in parallel have limited overlap. With this, they develop an algorithm that runs in polynomial work and polylogarithmic depth (i.e., it is in RNC) with an approximation ratio of $(1+\varepsilon)H_n$. The problem was also studied by Rajagopalan and Vazirani[RV98], who achieve better work-depth bounds but worse approximation ratio of about $2(1+\varepsilon)H_n$.

We abstract out a key component in their approaches, which we refer to as Maximal Near Independent Set (MaNIS). The definition of MaNIS is a natural generalization of Maximal Independent Set (MIS). In words, it offers a way to select a subcollection from a collection of sets and give a total order on this subcollection such that (1) adding sets in this order guarantees that each time a new set is added, at least a "large" fraction of the set's elements did not appear in the existing sets; and (2) the set not chosen in this subcollection has a significant overlap with the union of those in it.

Definition 1.4 (Ranked (ε, δ)-MaNIS) *Let $\varepsilon, \delta > 0$. Given a bipartite graph $G = (A \cup B, E)$, we say that a set $J = \{s_1, s_2, \dots, s_k\} \subseteq A$ is a* ranked (ε, δ) maximal nearly independent set*, or a ranked (ε, δ)-MaNIS for short, if*

(1) Nearly Independent. *There is an ordering (not part of the MaNIS solution)* $s_1, s_2, \ldots, s_k$ *such that each chosen option* s_i *is almost completely independent of* $s_1, s_2, \ldots, s_{i-1}$, *i.e., for all* $i = 1, \ldots, k$,

$$|N(s_i) \setminus N(\{s_1, s_2, \ldots, s_{i-1}\})| \geq (1 - \delta - \varepsilon)|N(s_i)|.$$

(2) Maximal. *The unchosen options have significant overlaps with the chosen options, i.e., for all* $a \in A \setminus J$,

$$|N(a) \setminus N(J)| < (1 - \varepsilon)|N(a)|.$$

We derive a simple $O(m)$ work and $O(\log^2 m)$ depth randomized algorithm for computing ranked $(\varepsilon, 3\varepsilon)$-MaNIS, where m is the number of edges in the bipartite graph G. With MaNIS, we develop a simple $O(m)$ work, $O(\log^3 m)$ depth randomized solution to set cover with an approximation ratio $(1 + \varepsilon)H_n$. This reduces the work by $O(\log^3 m)$ over the best previous known parallel (RNC) methods. In fact, the bucketing technique allows us to improve on the $\Theta(m \log m)$-work bound for the sequential *strictly* greedy algorithm, albeit at the loss of $(1+\varepsilon)$ in the approximation ratio. Furthermore, we apply the approach to solve several related problems in polylogarithmic depth, including a $(1 - 1/e - \varepsilon)$-approximation for max k-cover and a $(4+\varepsilon)$-approximation for min-sum set cover both in linear work; and an $O(\log^* n)$-approximation for asymmetric k-center for $k \leq \log^{O(1)} n$ and a $(1.861 + \varepsilon)$-approximation for metric facility location both in essentially the same work bounds as their sequential counterparts. These algorithms improve on previous results for all problems, and for asymmetric k-center and min-sum set cover are the first RNC algorithms that give a non-trivial approximation.

1.6 Low-Stretch Spanning Subtrees and Subgraphs, and Parallel SDD Solvers

Chapter 6 presents the design and analysis of a near linear-work parallel algorithm for solving symmetric diagonally dominant (SDD) linear systems. On input of a SDD n-by-n matrix A with m non-zero entries and a vector b, our algorithm computes a vector $\tilde{x}$ such that $\|\tilde{x} - A^+b\|_A \leq \varepsilon \cdot \|A^+b\|_A$ in $O(m \log^{O(1)} n \log \frac{1}{\epsilon})$ work and $O(m^{1/3+\theta} \log \frac{1}{\epsilon})$ depth for any fixed $\theta > 0$.

The algorithm relies on a parallel algorithm for generating low-stretch spanning trees or spanning subgraphs. To this end, we first develop a parallel decomposition algorithm that in polylogarithmic depth and $\widetilde{O}(|E|)$ work[3], partitions a graph into components with polylogarithmic diameter such that only a small fraction of the original edges are between the components. This can be used to generate low-stretch spanning trees with average stretch $O(n^{\alpha})$ in $O(n^{1+\alpha})$ work and $O(n^{\alpha})$ depth. Alternatively, it can be used to generate spanning subgraphs with polylogarithmic average stretch in $\widetilde{O}(|E|)$ work and polylogarithmic depth. We apply this subgraph construction to derive our solver. Specifically, we are able to prove the following theorem:

Theorem 1.5 *For any fixed $\theta > 0$ and any $\epsilon > 0$, there is an algorithm `SDDSolve` that on input an $n \times n$ SDD matrix A with m non-zero elements and a vector b, computes a vector $\tilde{x}$ such that $\|\tilde{x} - A^{+}b\|_A \leq \varepsilon \cdot \|A^{+}b\|_A$ in $O(m \log^{O(1)} n \log \frac{1}{\epsilon})$ work and $O(m^{1/3+\theta} \log \frac{1}{\epsilon})$ depth.*

By using the linear system solver in known applications, our results imply improved parallel randomized algorithms for several problems, including single-source shortest paths, maximum flow, min-cost flow, and approximate max-flow.

In the general solver framework of Spielman and Teng [ST06, KMP10], near linear-time SDD solvers rely on a suitable preconditioning chain of progressively smaller graphs. Assuming that we have an algorithm for generating low-stretch spanning trees, the algorithm as given in Koutis et al. [KMP10] parallelizes under the following modifications: (i) perform the partial Cholesky factorization in parallel and (ii) terminate the preconditioning chain with a graph that is of size approximately $m^{1/3}$. As discussed in more details later on, the construction of the preconditioning chain is a primary motivation of the main technical part of the work in this chapter, a parallel implementation of a modified version of Alon et al.'s low-stretch spanning tree algorithm [AKPW95].

More specifically, as a first step, we find a graph embedding into a spanning tree with average stretch $2^{O(\sqrt{\log n \log\log n})}$ in $\widetilde{O}(m)$ work and $O(2^{O(\sqrt{\log n \log\log n})} \log \Delta)$ depth, where Δ is the ratio of the largest to smallest distance in the graph. The original AKPW algorithm relies on a parallel graph decomposition scheme of Awerbuch [Awe85], which takes an unweighted graph and breaks it into components with a specified diameter and few crossing edges. While such schemes are known in the sequential setting, they do not parallelize readily because removing edges belonging to one component might increase the diameter or even disconnect subsequent components. We present the first near linear-work parallel decomposition algorithm that also gives strong-diameter guarantees. This leads a parallel solver algorithm.

[3]The $\widetilde{O}(\cdot)$ notion hides polylogarithmic factors.

1.7 Probabilistic Tree Embeddings and Applications

In Chapter 7, we present a parallel algorithm that embeds any n-point metric into a distribution of hierarchically well-separated trees (HSTs) with $O(n^2 \log n)$ (randomized) work and $O(\log^2 n)$ depth, while providing the same distance-preserving guarantees as that of Fakcharoenphol et al. (i.e., maintaining distances up to $O(\log n)$ in expectation) [FRT04]. Probabilistic tree embeddings—the general idea of embedding finite metrics into a distribution of dominating trees while maintaining distances in expectation—has proved to be an extremely useful and general technique in the algorithmic study of metric spaces [Bar98]. Their study has far-reaching consequences to understanding finite metrics and developing approximation algorithms on them. The particular algorithm we parallelize is an elegant optimal algorithm given by Fakcharoenphol, Rao, and Talwar (FRT).

The main challenge arises in making sure the depth of the computation is polylogarithmic even when the resulting tree is highly imbalanced—in contrast, the FRT algorithm, as stated, works level by level and can have high depth. This imbalance can occur when the ratio between the maximum and minimum distances in the metric space is large. Our contribution lies in recognizing an alternative view of the FRT algorithm and developing an efficient algorithm to exploit it. In addition, our analysis also implies probabilistic embeddings into trees without Steiner nodes of height $O(\log n)$ whp. (though not HSTs); such trees are useful for both our algorithms and have also proved useful in other contexts.

Using this algorithm, we give an RNC $O(\log k)$-approximation for k-median. This is the first RNC algorithm that gives non-trivial approximation for any k [4]. Specifically, we prove:

Theorem 1.6 *For $k \geq \log n$, the k-median problem admits a factor-$O(\log k)$ approximation with $O(nk + k(k\log(\frac{n}{k}))^2) \leq O(kn^2)$ work and $O(\log^2 n)$ depth. For $k < \log n$, the problem admits a $O(1)$-approximation with $O(n \log n + k^2 \log^5 n)$ work and $O(\log^2 n)$ depth.*

The algorithm is work efficient relative to previously described sequential techniques. We also give an RNC $O(\log n)$-approximation algorithm for buy-at-bulk network design. This algorithm is within an $O(\log n)$ factor of being work efficient.

Theorem 1.7 *The buy-at-bulk network design problem with k demand pairs on an n-node graph can be solved in $O(n^3 \log n)$ work and $O(\log^2 n)$ depth.*

[4]There is an RNC algorithm that give a $(5 + \varepsilon)$-approximation for $k \leq \text{polylog}(n)$ [BT10]

1.8 Hierarchical Diagonal Blocking

Investigated in Chapter 8, sparse matrix-vector multiplication (SpMV) lies at the heart of high performance parallel computing, from iterative numerical algorithms to shortest-path algorithms. On modern machine architectures, the performance of SpMV, however, is almost always limited by the system's memory bandwidth [WOV⁺07a, BKMT10]: the processors (cores) have more computing power than the memory system can keep pace with, resulting in the common finding that despite a substantial amount of parallelism available, the performance of SpMV algorithms does not scale beyond the first few cores. After performing a series of bandwidth studies, we identified the bottleneck: when computing the product $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, even if most of the vector's entries reside in cache, the memory system cannot supply the entries of $\boldsymbol{A}$ fast enough.

This finding suggests that we should be able to improve the performance of these SpMV routines if we can represent $\boldsymbol{A}$ with a smaller memory footprint. We present a matrix representation, called *Hierarchical Diagonal Blocking* (HDB), which captures many of the existing optimization techniques in a common representation. It can take advantage of symmetry while still being easy to parallelize. It takes advantage of reordering. It also allows for simple compression of column indices. In conjunction with precision reduction (storing single-precision numbers in place of doubles), it can reduce the overall bandwidth requirements by more than 3x. It is particularly well-suited for problems with symmetric matrices, for which the corresponding graphs have reasonably small graph separators, and for which the effects of reduced precision arithmetic are well-understood (combinatorial multigrid solvers are prime examples).

We give an SpMV algorithm for use with HDB representation, proving the following bounds in the parallel cache oblivious model:

Theorem 1.8 *Let $\mathcal{M}$ be a class of matrices for which the adjacency graphs satisfy an n^α-edge separator theorem, $\alpha < 1$, and $\boldsymbol{A} \in \mathcal{M}$ be an n-by-n matrix with $m \geq n$ nonzeros, or $m \geq n$ lower triangular nonzeros for a symmetric matrix. If $\boldsymbol{A}$ is stored in the HDB representation T then, on a machine with word size w:*

(1) T can be implemented to use $m + O(n/w)$ words.

(2) There is a cache oblivious and runs with $m/B + O(1 + n/(Bw) + n/M^{1-\alpha})$ misses in the ideal cache model. The algorithm runs in $O(\log^{O(1)} n)$ depth.

The key idea of this representation is as follows: if the graph (corresponding to the matrix) satisfies the n^α-edge separator theorem, then there is a reordering that makes the matrix entries look like blocks around the diagonal, without too many edges going across the blocks. This holds at multiple scales, forming a hierarchy of block structures, which has several advantages. For instance, since these blocks are non-interfering, they can be executed in parallel. Furthermore, within a block, we can perform index compression, using fewer bits to represent the column indices.

We complement the theoretical results with a number of experiments that evaluate the performance of various SpMV schemes on recent multicore architectures. The results show that by reducing the bandwidth requirements, we not only enjoy substantial performance gain, but are able to scale much better on multiple cores when the bandwidth becomes more limiting. Our results show that a simple double-precision parallel SpMV algorithm saturates the multicore bandwidth, but by reducing the bandwidth requirements—using a combination of hierarchical diagonal blocking and precision reduction—on an 8-core Nehalem machine, we are able to obtain, on average, *a* $2.5x$ *speedup over the simple parallel implementation.*

1.9 Parallel and I/O Efficient Set Cover and Related Problems

Chapter 9 presents the design, analysis, and implementation of parallel and sequential I/O-efficient algorithms for set cover, max cover, and related problems, tying together the line of work on parallel set cover and the line of work on efficient set cover algorithms for large, disk-resident instances.

We design and analyze a parallel cache-oblivious set-cover algorithm that offers essentially the same approximation guarantees as the standard greedy algorithm. This algorithm is the first efficient external-memory or cache-oblivious algorithm for when neither the sets nor the elements fit in memory, leading to I/O cost (cache complexity) equivalent to sorting. The algorithm also implies low cache-misses on parallel hierarchical memories (again, equivalent to sorting). More specifically, we prove the following theorem:

Theorem 1.9 (Parallel and I/O Efficient Set Cover) *The I/O (cache) complexity of the approximate set cover algorithm on an instance of size W is $O(sort(W))$ and the depth is polylogarithmic in W. Furthermore, this implies an algorithm for prefix-optimal max cover and min-sum set cover in the same complexity bounds.*

Building on this theory, our main contribution is the implementation of slight variants of the theoretical algorithm optimized for different hardware setups. We provide extensive experimental evaluation showing non-trivial speedups over existing algorithms without compromising the solution's quality.

1.10 Bibliographical and Personal Notes

This thesis is a compendium of the end product resulting from extensive collaboration with various people over the years. Chapter 3 on facility-location problems is joint work with Guy Blelloch and has previously appeared in SPAA'10 [BT10]. The results in this chapter represent the starting point of this thesis, convincing us that parallelizing these algorithms requires new ideas and is an important and interesting pursuit; the supplemented k-median proof was extracted from joint work with Anupam Gupta [GT08]. The results on MaxCut in Chapter 4 stem from Avrim Blum's questions after my theory lunch talk and have benefited greatly from lectures presented in the parallel approximation algorithms reading group (PAARG) and conversations with Guy Blelloch, Anupam Gupta, and Richard Peng. The body of work in Chapter 5 on Maximal Nearly Independent Set (MaNIS) is joint work with Guy Blelloch and Richard Peng; the definition of MaNIS and the presentation of the MaNIS algorithm have been greatly simplified after various illuminating discussions with Anupam Gupta. In addition, Gary Miller made us think deeper about the underlying connection between MIS and MaNIS algorithms, as well as their proof techniques, which helped shape the current proof. The work previously appeared in SPAA'11 [BPT11].

Chapter 6 on low-stretch subgraphs and parallel SDD solvers is my biggest collaboration project to date, expanding on the conference version which originally appeared in SPAA'11 [BGK$^+$11]. This work is joint with Guy Blelloch, Anupam Gupta, Ioannis Koutis, Gary Miller, and Richard Peng. Chapter 7 on probabilistic tree embeddings and applications results from collaboration with Guy Blelloch and Anupam Gupta. Chapter 8 on hierarchical diagonal blocking is based on my paper with Guy Blelloch, Ioannis Koutis and Gary Miller in SC'10 [BKMT10]. The SpMV code was originally developed as part of the Parallel Benchmark Suite, and the integration was motivated by Ioannis's prior experience with using Intel MKL in his combinatorial multigrid solver. Finally, Chapter 9 on I/O efficient set cover and related algorithms is joint work with Guy Blelloch and Harsha Simhadri.

Chapter 2

Preliminaries and Notation

Throughout this document, we denote by $[k]$ the set $\{1, 2, \ldots, k\}$ and use the notation $\widetilde{O}(f(n))$ to hide polylog and polyloglog factors, i.e., $\widetilde{O}(f(n))$ means $O(f(n)\,\mathrm{polylog}(f(n)))$. We say that an event happens *with high probability (w.h.p.)* if it happens with probability exceeding $1 - n^{-\Omega(1)}$.

For a graph G, we denote by $\deg_G(v)$ the degree of the vertex v in G and use $N_G(v)$ to denote the neighbor set of the node v. Furthermore, we write $u \sim v$ for u is adjacent to v. Extending this notation, we write $N_G(X)$ to mean the neighbors of the vertex set X, i.e., $N_G(X) = \cup_{w \in X} N_G(w)$. For a vertex set $S \subseteq V$, we write $\overline{S}$ to denote the complement of S, i.e., $\overline{S} = V \setminus S$. Furthermore, we denote by $E(S, \overline{S})$ the edges crossing the cut S and $\overline{S}$, that is, $E(S, \overline{S}) = \{uv \in E : u \in S, v \in \overline{S}\}$. We drop the subscript (i.e., writing $\deg(v)$, $N(v)$, and $N(X)$) when the context is clear. Let $V(G)$ and $E(G)$ denote respectively the set of nodes and the set of edges of the graph G.

Metric Space. Let X be a set and d: $X \times X \to \mathbb{R}_+ \cup \{0\}$ be a distance function on X. We say that (X, d) is a metric space if

(i) $d(x, y) = 0$ iff. $x = y$,
(ii) $d(x, y) = d(y, x)$ for all $x, y \in X$, and
(iii) $d(x, y) \leq d(x, z) + d(z, y)$ for all $x, y, z \in X$.

2.1 Parallel Models and Performance of Parallel Algorithms

Several parallel computing models have been proposed in the literature. Throughout this work, our low-level (multiprocessor) model of choice is a synchronous shared-memory model, commonly known as the parallel random-access machine (PRAM), a generalization of the RAM model for sequential computing [JáJ92]. The PRAM model has the advantage of a simple, formal model with well-understood connections to other models of parallel machines. Within the PRAM model, there are variations depending on how concurrent operations are handled. Standard variants include exclusive-read exclusive-write (EREW), concurrent-read exclusive-write (CREW) and concurrent-read concurrent-write (CRCW). Within concurrent-write models, there are different variations (e.g., arbitrary write, and maximum-priority write) depending how write conflicts are handled. These models were to shown to have similar expressivity and power, with EREW being the most restrictive. In this work, we take the liberty of choosing the model that eases the presentation of a particular algorithm and discuss its implications in related models.

However, at the level of PRAM abstraction, an algorithm's description still involves red-herring details. Because of the nature of our algorithms, we prefer a higher level of abstraction, an abstraction with a primary focus on algorithms and less so on the specifics of how they are executed. For this, we adopt the work-depth model [JáJ92]. In the *work-depth* model, the performance of an algorithm is determined by examining two important parameters: *work* (W)—the total number of operations—and *depth* (D), also known as *span*—the dependencies among the operations. It is also common to refer to work as T_1 (time on one processor) and depth as T_∞ (time on infinitely many processors).

More specifically, our algorithms are presented in the nested parallel model, allowing arbitrary dynamic nesting of parallel loops and fork-join constructs but no other synchronizations (all standard textbook parallel algorithms can be represented in this model). This corresponds to the class of algorithms with series-parallel dependence graphs (see Figure 2.1). Computations can be decomposed into "tasks", "parallel blocks" and "strands" recursively: As a base case, a *strand* is a serial sequence of instructions not containing any parallel constructs or subtasks. A *task* is formed by serially composing $k \geq 1$ strands interleaved with $(k-1)$ "parallel blocks" (denoted by $\mathsf{t} = \mathsf{s}_1; \mathsf{b}_1; \ldots; \mathsf{s}_k$). A *parallel block* is formed by composing in parallel one or more tasks with a fork point before all of them and a join point after (denoted by $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$). A parallel block can be, for example, a parallel loop or some constant number of recursive calls. The top-level computation is a task. The depth (aka. span) of a computation is therefore the length of the longest path in the dependence graph.

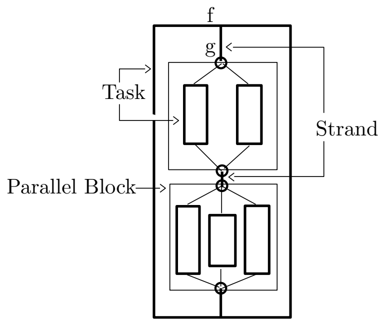


Figure 2.1: Decomposing the computation: tasks, strands and parallel blocks

The following theorem shows a relationship between the work-depth model and CREW PRAM:

Theorem 2.1 (Brent's Theorem [Bre74, JáJ92]) *An algorithm with work W and depth D can be executed on a CREW PRAM machine with p processors in $O(W/p + D)$ time, using a greedy scheduler.*

2.1.1 Parallel Cache-Oblivious Model

On a real computer system, the performance of an algorithm can depend heavily on its cache performance. The cache-oblivious approach for analyzing algorithms offers a framework for analyzing the cache cost on a simple, single-level cache, so that the resulting analysis can be used to imply good performance bounds on a variety of hierarchical caches [FLPR99]. The *ideal-cache model* is used for analyzing cache costs. It is a two-level model of computation consisting of an unbounded memory and a cache of size M. Data are transferred between the two levels using cache lines of size B; all computation occurs on data in the cache. The model can run any standard computation designed for a random access machine (RAM model) on words of memory, and the cost is measured in terms of the number of misses incurred by the computation. This cost, often denoted by $Q(C; M, B)$, is referred to as the cache complexity for a computation C. Often, we write $Q\,(n; M, B)$ for $Q(C; M, B)$ when the input has size n and the computation is clear from the context.

The model assumes an ideal cache that evicts the location that will be needed the furthest into the future (this is the optimal policy). In reality, no real caches are ideal or can even be ideal since future accesses are not known at the time of eviction. However, such an ideal cache implies certain bounds on more realistic cache models. For example, an ideal cache complexity can be applied on a fully-associative LRU (least recently used) cache with at most a factor of 2 increase in misses and cache size [FKL+91]. Furthermore, an ideal cache can be simulated on set-associative caches.

An algorithm is *cache oblivious* in the ideal-cache model if it does not take into account the size of M or B (or any other features of the cache). If a cache oblivious algorithm A has cache complexity $Q(A; B, M)$ on a machine with block size B and cache size M, then on a hierarchical cache with cache parameters (M_i, B_i) at level i, the algorithm will suffer at most $Q(A; M_i, B_i)$ misses at each level i. Therefore, if $Q(A; M_i, B_i)$ is asymptotically optimal for B and M, it is optimal for all levels of the cache.

This type of analysis has recently been extended to the parallel setting [BGS10, BFGS11]. For nested parallel computations, one can analyze the algorithm using a sequential ordering and then use general results to bound cache misses on parallel machines with either shared or private hierarchical caches. This works well when the algorithms have low depth. In particular, for a shared-memory parallel machine with private caches (each processor has its own cache) using a work-stealing scheduler, $Q_p(A; M, B) < Q(A; M, B) + O(pMD/B)$ with probability $1 - \delta$ [ABB02], and for a shared cache using a parallel-depth-first (PDF) scheduler, $Q_p(A; M + pBD, B) \leq Q(A; M, B)$ [BG04], where D is the depth of the computation and p the number of processors. This can be formalized and extended to a more general setting (i.e., ireggular computation) as follows.

The Parallel Cache-Oblivious model is a simple, high-level model for algorithm analysis. Like the cache-oblivious model, in the *Parallel Cache-Oblivious (PCO) model*, there is a memory of unbounded size and a single cache with size M, line-size B (in words), and optimal (i.e., furthest into the future) replacement policy. The cache state κ consists of the set of cache lines resident in the cache at a given time. When a location in a non-resident line l is accessed and the cache is full, l replaces in κ the line accessed furthest into the future, incurring a *cache miss*.

To extend the CO model to parallel computations, one needs to define how to analyze the number of cache misses during the execution of a parallel block. The PCO model approaches it by (i) ignoring any data reuse among the subtasks and (ii) flushing the cache at each fork and join point of any task that does not fit within the cache, as follows. Let $loc(\mathsf{t}; B)$ denote the set of distinct cache lines accessed by task t, and $S(\mathsf{t}; B) = |loc(\mathsf{t}; B)| \cdot B$ denote its size (also let $s(\mathsf{t}; B) = |loc(\mathsf{t}; B)|$ denote the size in terms of number of cache lines). Let $Q(\mathsf{c}; M, B; \kappa)$ be the cache complexity of c in the sequential CO model when starting with cache state κ.

Task t forks subtasks t_1 and t_2,
with $\kappa = \{l_1, l_2, l_3\}$

t_1 accesses l_1, l_4, l_5 incurring 2 misses
t_2 accesses l_2, l_4, l_6 incurring 2 misses

At the join point: $\kappa' = \{l_1, l_2, l_3, l_4, l_5, l_6\}$

Figure 2.2: Applying the PCO model (Definition 2.2) to a parallel block. Here, $Q^*(\mathsf{t}; M, B; \kappa) = 4$.

Definition 2.2 (Parallel Cache-Oblivious Model) *For cache parameters M and B the cache complexity of a strand, parallel block, and a task starting in cache state κ are defined recursively as follows (refer to [BFGS11] for more details). For a strand, $Q^*(\mathsf{s}; M, B; \kappa) = Q(\mathsf{s}; M, B; \kappa)$. For a parallel block, $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$,*

$$Q^*(\mathsf{b}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{t}_i; M, B; \kappa).$$

For a task $\mathsf{t} = \mathsf{c}_1; \mathsf{c}_2; \ldots; \mathsf{c}_k$,

$$Q^*(\mathsf{t}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{c}_i; M, B; \kappa_{i-1}),$$

where $\kappa_i = \emptyset$ if $S(\mathsf{t}; B) > M$, and

$$\kappa_i = \kappa \cup_{j=1}^{i} loc(\mathsf{c}_j; B)$$

if $S(\mathsf{t}; B) \leq M$.

We use $Q^*(\mathsf{c}; M, B)$ to denote a computation c starting with an empty cache, $Q^*(n; M, B)$ when n is a parameter of the computation. We note that $Q^*(\mathsf{c}; M, B) \geq Q(\mathsf{c}; M, B)$. When applied to a parallel machine Q^* is a "work-like" measure and represents the total number of cache misses across all processors. An appropriate scheduler is used to evenly balance them across the processors.

As a general guideline: being able to design an algorithm with reasonably low depth is a big win as the bounds indicate. We note that recursive matrix multiplication, FFT, Barnes-Hut n-body code, merging, mergesort, quicksort, k-nearest neighbors, direct solvers using nested dissection, all have low depth and are reasonably efficient under the cache oblivious model.

2.2 Graph Separators

Informally, a graph has $n^\alpha, \alpha < 1$ edge separators if there is a cut that partitions the graph into two almost equal-sized parts such that the number of edges between the two parts is no more than n^α, within a constant. To properly deal with asymptotics and what it means to be "within a constant," separators are typically defined with respect to an infinite class of graphs. Formally, let $\mathcal{S}$ be a class of graphs that is closed under the subgraph relation (i.e., for $G \in \mathcal{S}$, every subgraph of G is also in $\mathcal{S}$). We say that $\mathcal{S}$ satisfies an $f(n)$-*edge separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that every graph $G = (V, E)$ in $\mathcal{S}$ with n vertices can be partitioned into two sets of vertices V_a, V_b such that

$$\mathsf{cutSize}(V_a, V_b) \quad \overset{def}{:=} \quad |E \cap (V_a \times V_b)| \quad \leq \quad \beta f(n),$$

where $|V_a|, |V_b| \leq \alpha n$ [LT79]. It is well-known that bounded-degree planar graphs and graphs with bounded genus satisfy an $n^{1/2}$-edge separator theorem. It is also known that certain well-shaped meshes in d dimensions satisfy a $n^{(d-1)/d}$-edge separator theorem [MTV91]. We note that such meshes allow for features that vary in size by large factors (e.g. small near a singularity and large where nothing is happening), but require a smooth transition from small features to large features. In addition, many other types of real-world graphs have good separators, including, for example, a link graph from Google [BBK04].

Edge separators are often applied recursively to generate a separator tree with the vertices at the leaves and the cuts at internal nodes. Such a separator tree can be used to reorder the vertices based on an in- or post- order traversal of the tree. It is not hard to show that for graphs satisfying an n^α separator theorem, the tree can be fully balanced while maintaining the $O(n^\alpha)$ separator sizes at each node.

Separators have been used for many applications. The seminal work of Lipton and Tarjan showed how separators can be used in nested dissection to generate efficient direct solvers [LT79]. Another common application is to partition data structures across parallel machines to minimize communication. It has also been used to compress graphs [BBK03] down to a linear number of bits. The idea is that if the graph is reordered using separators, then most of the edges are "short" and can thus be encoded using fewer bits than other edges.

2.3 Existing Techniques in Parallel Approximation Algorithms

There are a number of techniques we can learn from RNC and NC algorithms that have been developed over the years. These techniques provide invaluable lessons in developing the results in this thesis.

Geometric-Scaling Approximation. To create opportunities for parallelism, many algorithms resort to bulk processing by grouping similar items together. A common idea is that instead of picking only the "best" option, they make room for parallelism by allowing a small slack (e.g., a $(1+\varepsilon)$ factor) in what can be selected. Often used in conjunction with a filter step, this allows multiple options to be selected together in bulk, which reduces the depth. This idea often results in a slightly worse approximation factor than the sequential version. For instance, the parallel set-cover algorithm of Rajagopalan and Vazirani is a $(2(1+\varepsilon)\ln n)$-approximation, compared to a $\ln n$-approximation produced by the standard greedy set cover. Likewise, the parallel vertex-cover algorithm of Khuller et al. is a $2/(1-\varepsilon)$-approximation as opposed to the optimal 2-approximation given by various known sequential algorithms. Only recently has the approximation factor for vertex cover been improved to 2 in the parallel case [KY09], which avoids the geometric-scaling approximation. In general, geometric scaling makes the number of groups that have to be processed sequentially a logarithmic function of the ratio between the most costly option and the least inexpensive option under some valuation.

Subselecting Nearly Non-Overlapping Set. Many optimization problems deal with sets, and one of the major obstacles in parallelizing algorithms for these problems is in ensuring that the sets that the algorithm ends up choosing do not have significant overlap with each other. A prime example is the situation that happens in the set cover problem, which was our initial motivation for the work on Maximal Nearly-Independent Sets (MaNIS) in Chapter 5. Applying a geometric-scaling approximation to set cover creates the following scenario: faced with a number of sets of roughly equal size (i.e., between, say, $(1-\varepsilon)s$ and s), we want to choose some of these sets to ensure that the selected sets taken together cover approximately as many elements as the sum of the individual set sizes.

Berger, Rompel, and Shor [BRS94] developed a clever technique, which we call independent sampling, that takes care of this situation. This sampling technique should be compared with Luby's Maximal Independent Set (MIS) algorithm that flips a coin for each vertex. Here the idea is that if we choose a set with small enough probability, proportional to how much overlap there is but independently for all sets, the chosen sets are unlikely to "collide" in too many places. Later, Rajagopalan and Vazirani [RV98] proposed a different technique, which we call permutation sampling. This bears much similarity to Luby's MIS algorithm where a random number is picked for each node. The key idea in this case is to select a set if the set "wins" most of its elements after assigning each element to the set which the element belongs to that has the highest ranking in the permutation order. The work in Chapter 5 formalizes, generalizes, and simplifies the crucial steps in these results.

Linear Programs Approximation. Linear programs (LPs) play an important role in the design and analysis of sequential approximation algorithms, in part because there are fast algorithms for solving LPs. The problem in the parallel setting is that none of these algorithms have small depth. In fact, solving a general LP or even approximating the optimal value to any constant factor is P-complete, making it unlikely that an (R)NC algorithm is possible. There is, however, a class of LPs that can be approximated to any constant ε accuracy in RNC.

Positive linear programs (PLPs) are a class of linear programs in which all coefficients both in the objective and the constraints, if non-zero, are positive. The most general form, also known as *mixed covering and packing* programs, is either minimizing or maximizing $c_1x_1 + c_2x_2 + \cdots + c_nx_n$ subject to $Cx \geq b$ and $Px \leq d$, where we require $x \in \mathbb{R}^n_+$, $c_i \geq 0$, $C_{i,j} \geq 0$, $D_{i,j} \geq 0$, $b_i > 0$ and $d_i > 0$. The constraints $Cx \geq b$ are covering constraints and $Px \leq d$, packing constraints.

For the covering version (or its dual packing version), Luby and Nisan [LN93] presented an NC $O(\frac{1}{\varepsilon^4} \log n \log N \log(m/\varepsilon))$-depth $O(\frac{1}{\varepsilon^4} N \log n \log N \log(m/\varepsilon))$-work algorithm, where n is the number of variables, m the number of constraints, and N the number of non-zero coefficients in the matrix C. This result implies that we can obtain an $O(\log n)$-approximation to set cover in $\widetilde{O}(N)$, where N here is the sum of the set sizes. But this would not be work efficient relative the sequential greedy algorithm.

Subsequently, Young [You01] presented a parallel algorithm that can approximate programs that are both covering and packing (aka. mixed covering and packing programs) in essentially the same work-depth bounds—but the dependence on ε was still $1/\varepsilon^4$.

SDPs Approximation. Very recently, Jain and Yao [JY11] announced a parallel algorithm for approximating positive SDPs up to an arbitrary constant accuracy in polylogarithm depth and poly(N) work. This directly implies an SDP-based algorithm for MaxCut; however, the generality of this approach comes at a cost. We believe the MaxCut algorithm derived via this route does significantly more work than our algorithm for MaxCut from Chapter 4.

Chapter 3

Facility-Location Problems

Facility location is an important and well-studied class of problems in approximation algorithms, with far-reaching implications in areas as diverse as machine learning, operations research, and networking: the popular k-means clustering and many network-design problems are all examples of problems in this class. Not only are these problems important because of their practical value, but they appeal to study because of their special stature as “testbeds” for techniques in approximation algorithms. Recent research has focused primarily on improving the approximation guarantee, producing a series of beautiful results, some of which are highly efficient—often, with the sequential running time within constant or polylogarithmic factors of the input size.

Despite significant progress on these fronts, work on developing parallel approximation algorithms for these problems remains virtually non-existent. Although variants of these problems have been considered in the the distributed computing setting [MW05, GLS06, PP09], to the best our of knowledge, almost no prior work has looked directly in the parallel setting where the total work and parallel time (depth) are the parameters of concern. The only prior work on these problems is due to Wang and Cheng, who gave a 2-approximation algorithm for k-center that runs in $O(n \log^2 n)$ depth and $O(n^3)$ work [WC90], a result which we improve upon in this work.

Deriving parallel algorithms for facility location problems is a non-trivial task and will be a valuable step in understanding how common techniques in approximation algorithms can be parallelized efficiently. Previous work on facility location commonly relies on techniques such as linear-program (LP) rounding, local search, primal dual, and greedy. Unfortunately, LP rounding relies on solving a class of linear programs not known to be solvable efficiently in polylogarithmic time. Neither do known techniques allow for parallelizing local-search algorithms. Despite some success in parallelizing primal-dual and greedy algorithms for set-covering, vertex-covering, and related problems, these algorithm are obtained using problem-specific techniques, which are not readily applicable to other problems.

Summary of Results

In this chapter, we explore the design and analysis of several algorithms for (metric) facility location, k-median, k-means and k-center problems, focusing on parallelizing a diverse set of techniques in approximation algorithms. We study the algorithms on the EREW PRAM and the Parallel Cache Oblivious model [BGS10, BFGS11]. The latter model is a generalization of the cache-oblivious model, which captures memory locality. We are primarily concerned with minimizing the work (or cache complexity) while achieving polylogarithmic depth in these models. We are less concerned with polylogarithmic factors in the depth since such measures are not robust across models. By work, we mean the total operation count. All algorithms we develop are in NC or RNC, so they have polylogarithmic depth.

We first present a parallel algorithm mimicking the greedy algorithm of Jain et al. [JMM+03]. This is the most challenging algorithm to parallelize because the greedy algorithm is inherently sequential. We show the algorithm gives a $(6+\varepsilon)$-approximation and does $O(m\log^2_{1+\varepsilon} m)$ work, which is within a logarithmic factor of the serial algorithm. Then, we present a simple RNC algorithm using the primal-dual approach of Jain and Vazirani [JV01b] which leads to a $(3+\varepsilon)$-approximation and for input of size m runs in $O(m\log_{1+\varepsilon} m)$ work, which is the same as the sequential work. The sequential algorithm is a 3-approximation. Following that, we present a local-search algorithm for k-median and k-means, with approximation factors of $5+\varepsilon$ and $81+\varepsilon$, matching the guarantees of the sequential algorithms. For constant k, the algorithm does $O(n^2\log n)$ work, which is the same as the sequential counterpart. Furthermore, we present a 2-approximation algorithm for k-center with $O((n\log n)^2)$ work, based on the algorithm of Hochbaum and Shmoys [HS85]. Finally, we show a $O(m\log^2_{1+\varepsilon}(m))$-work randomized rounding algorithm, which yields a $(4+\varepsilon)$-approximation, given an optimal linear-program solution as input. The last two algorithms run in work within a logarithmic factor of the serial algorithm counterparts.

Remarks: The greedy parallel facility location has since been improved to a $(3+\varepsilon)$-approximation, with the introduction of a new technique for resolving conflicts between competing options. This improved result is presented Chapter 5.

3.0.1 Related Work

Facility-location problems have had a long history. Because of space consideration, we mention only some of the results here, focusing on those concerning metric instances. For the (uncapacitated) metric facility location, the first constant factor approximation was given by Shmoys et al. [STA97], using an LP-rounding technique, which has subsequently been improved [Chu98, GK99]. A different approach, based on local-search techniques, has been used to obtain a 3-approximation [KPR00, AGK+04, GT08]. Combinatorial algorithms based on primal-dual and greedy approaches with constant approximation factors are also known [JMM+03, JV01b, PT03]. Other approximation algorithms and hardness results have also been given by [Svi02, CS03, Byr07, CG05, MMSV01, MYZ02, KPR00, CG05, GK99]. An open problem is to close the gap between the best known approximation factor of 1.5 [Byr07] and the hardness result of 1.463 [GK99].

At FOCS'99, the first constant factor approximation for k-median problem was presented by Charikar et al. [CGTS02], which was subsequently improved by [CG05] and [AGK+04] to the current best factor of $3+\varepsilon$. For k-means, constant-factor approximations are known for this problem [JV01b, GT08]; a special case when the metric space is the Euclidean space has also been studied [KMN+04]. For k-center, tight bounds are known: there is a 2-approximation algorithm due to [Gon85, HS86], and this is tight unless $\mathsf{P} = \mathsf{NP}$.

3.1 Preliminaries and Notation

Let F denote a set of *facilities* and C denote a set of *clients*. For convenience, let $n_c = |C|$, $n_f = |F|$, and $m = n_c \times n_f$. Each facility $i \in F$ has a cost of f_i, and each client $j \in C$ incurs a cost ("distance") $d(j, i)$ to use the facility i. We assume throughout that there is a metric space (X, d) with $F \cup C \subseteq X$ that underlies our problem instances. Thus, the distance d is symmetric and satisfies the triangle inequality. As a shorthand, denote the cost of the optimal solution by opt, the facility set of the optimal solution by F^*, and the facility set produced by our algorithm by F_A. Furthermore, we write $d(u, S)$ to mean the minimum distance from u to a member of S, i.e., $d(u, S) = \min\{d(u, w) : w \in S\}$.

Parallel Primitives. All the parallel algorithms in this chapter can be expressed in terms of a set of simple operations on vectors and dense matrices, making it easy to analyze costs on a variety of parallel models. In particular, the distances $d(\cdot, \cdot)$ can be represented as a dense $n \times n$ matrix, where $n = n_c + n_f$, and any data at clients or facilities can be represented as vectors. The only operations we need are parallel loops over the elements of the vector or matrix, transposing the matrix, sorting the rows of a matrix, and summation, prefix sums and distribution across the rows or columns of a matrix or vector. A prefix sum returns to each element of a sequence the sum of previous elements. The summation or prefix sum needs to be applied using a variety of associative operators, including min, max, and addition.

We refer to all the operations other than sorting as the *basic matrix operation*. The basic matrix operations on m elements can all be implemented with $O(m)$ work and $O(\log m)$ time on the EREW PRAM [JáJ92], and with $O(m/B)$ cache complexity and $O(\log m)$ depth in the parallel cache oblivious model. For the parallel cache oblivious model, we assume a tall cache $M > B^2$, where M is the size of the cache and B is the block size. Sorting m elements takes $O(m \log m)$ work and $O(\log m)$ time on an EREW PRAM [Col88], and $O(\frac{m}{B} \log_{M/B} m)$ cache complexity and $O(\log^2 m)$ depth on the parallel cache oblivious model [BGS10]. All algorithms described in this chapter are *cache efficient* in the sense that the cache complexity in the cache oblivious model is bounded by $O(w/B)$ where w is the work in the EREW model. All algorithms use a polylogarithmic number of calls to the basic matrix operations and sorting and are thus in RNC—do polynomial work with polylogarithmic depth and possibly use randomization.

Given this set up, the problems considered in this chapter can be defined as follows:

Facility Location. The goal of this problem is to find a set of facilities $F_S \subseteq F$ that minimizes the objective function

$$\textsc{FacLoc}(F_S) = \sum_{i \in F_S} f_i + \sum_{j \in C} d(j, F_S) \tag{3.1}$$

Note that we do not need an explicit client-to-facility assignment because given a set of facilities F_S, the cost is minimized by assigning each client to the closest open facility.

Non-trivial upper- and lower-bounds for the cost of the optimal solution are useful objects in approximation algorithms. For each client $j \in C$, let $\gamma_j = \min_{i \in F}(f_i + d(j, i))$ and $\gamma = \max_{j \in C} \gamma_j$. The following bounds can be easily established:

$$\gamma \leq \mathsf{opt} \leq \sum_{j \in C} \gamma_j \leq \gamma n_c. \tag{3.2}$$

Furthermore, metric facility location has a natural integer-program formulation for which the relaxation yields the pair of primal and dual programs shown in Figure 3.1.

k-**Median and** k-**Means.** Unlike facility location, the k-median objective does not take into consideration facility costs, instead limiting the number of opened centers (facilities) to k.

$$\textbf{Minimize} \quad \sum_{i \in F, j \in C} d(j,i)x_{ij} \; + \; \sum_{i \in F} f_i y_i$$

$$\textbf{Subj. to:} \begin{cases} \sum_{i \in F} x_{ij} & \geq & 1 & \text{for } j \in C \\ y_i - x_{ij} & \geq & 0 & \text{for } i \in F, j \in C \\ x_{ij} \geq 0, \; y_i \geq 0 \end{cases} \qquad \text{(Primal LP)}$$

$$\textbf{Maximize} \quad \sum_{j \in C} \alpha_j$$

$$\textbf{Subj. to:} \begin{cases} \sum_{j \in C} \beta_{ij} & \leq & f_i & \text{for } i \in F \\ \alpha_j - \beta_{ij} & \leq & d(j,i) & \text{for } i \in F, j \in C \\ \beta_{ij} \geq 0, \; \alpha_j \geq 0 \end{cases} \qquad \text{(Dual DP)}$$

Figure 3.1: The primal and dual programs for metric (uncapacitated) facility location.

Moreover, in these problems, we typically do not distinguish between facilities and clients; every node is a client, and every node can be a facility. Formally, let $V \subseteq X$ be the set of nodes, and the goal is to find a set of at most k centers $F_S \subseteq V$ that minimizes the objective $\text{kMed}(F_S) = \sum_{j \in V} d(j, F_S)$. Almost identical to k-median is the k-means problem with the objective $\text{kMeans}(F_S) = \sum_{j \in C} d^2(j, F_S)$.

k-**Center.** Another type of facility-location problem which has a hard limit on the number of facilities to open is k-center. The k-center problem is to find a set of at most k centers $F_S \subseteq V$ that minimizes the objective $\text{kCenter}(F_S) = \max_{j \in V} d(j, F_S)$. In these problems, we will use n to denote the size of V.

3.2 Dominator Set

We describe two variants of the maximal independent set (MIS) problem, which will prove to be useful in nearly all algorithms described in this work. The first variant, called the *dominator set* problem, concerns finding a maximal set $I \subseteq V$ of nodes from a simple graph $G = (V, E)$ such that none of these nodes share a common neighbor (neighboring nodes of G cannot both be selected). The second variant, called the U*-dominator set* problem, involves finding a maximal set $I \subseteq U$ of the U-side nodes of a bipartite graph $H = (U, V, E)$ such that none of the nodes have a common V-side neighbor. We denote by $\text{MaxDom}(G)$ and $\text{MaxUDom}(H)$ the solutions to these problems, resp.

Both variants can be equivalently formulated in terms of maximal independent set. The first variant amounts to finding a maximal independent set on

$$G^2 = (V, \{uw : uw \in E \text{ or } \exists z \text{ s.t. } uz, zw \in E\}),$$

and the second variant a maximal independent set on

$$H' = (U, \{uw : \exists z \in V \text{ s.t. } uz, zw \in E\}).$$

Because of this relationship, on the surface, it may seem that one could simply compute G^2 or H' and run an existing MIS algorithm. Unfortunately, computing graphs such as G^2 and H' appears to need $O(n^\omega)$ work, where ω is the matrix-multiply constant, whereas the naïve greedy-like sequential algorithms for the same problems run in $O(|E|) = O(n^2)$. This difference makes it unlikely to obtain work efficient algorithms via this route.

In this section, we develop near work-efficient algorithms for these problems, bypassing the construction of the intermediate graphs. The key idea is to compute a maximal independent set in-place. Numerous parallel algorithms are known for maximal independent set, but the most relevant to us is an algorithm of Luby [Lub86], which we now sketch.

The input to the algorithm is a graph $G = (V, E)$. Luby's algorithm constructs a maximal independent set $I \subseteq V$ by proceeding in multiple rounds, with each round performing the following computation:

Algorithm 3.2.1 The select step of Luby's algorithm for maximal independent set.

```
1. For each $i \in V$, **in parallel**, $\pi(i)$ = a number chosen u.a.r. from $\{1, 2, \ldots, 2n^4\}$.
2. Include a node $i$ in the maximal independent set $I$ if $\pi(i) < \min\{\pi(j) : j \in N(i)\}$,
   where $N(i)$ is the neighborhood of $i$ in $G$.
```

This process is termed the *select step* in Luby's work. Following the select step, the newly selected nodes, together with their neighbors, are removed from the graph before moving on to the next round.

Implementing the select step: We describe how the select step can be performed in-place for the first variant; the technique applies to the other variant. We will be simulating running Luby's algorithm on G^2, without generating G^2. Since G^2 has the same node set as G, step 1 of Algorithm 3.2.1 remains unchanged. Thus, the crucial computation for the select step is to determine efficiently, for each node i, whether $\pi(i)$ holds the smallest number among its neighbors in G^2, i.e., computing efficiently the test in step 2. To accomplish this, we simply pass the $\pi(i)$ to their neighbors taking a minimum, and then to the neighbors again taking a minimum. These can be implemented with a constant number of basic matrix operations, in particular distribution and summation using minimum over the rows and columns of the $|V|^2$ matrix.

Lemma 3.1 *Given a graph $G = (V, E)$, a maximal dominator set $I \subseteq V$ can be found in expected $O(\log^2 |V|)$ depth and $O(|V|^2 \log |V|)$ work. Furthermore, given a bipartite graph $G = (U, V, E)$, a maximal U-dominator set $I \subseteq U$ can be found in expected $O((\log |U|) \cdot \max\{\log |U|, \log |V|\})$ depth and $O(|V||U| \max\{\log |U|, \log |V|\})$ work.*

We note that for sparse matrices, which we do not use in this chapter, this can be easily improved to $O(|E| \log |V|)$ work.

3.3 Facility Location: Greedy

The greedy scheme underlies an exceptionally simple algorithm for facility location, due to Jain et al. [JMM+03]. Despite the simplicity, the algorithm offers one of the best known approximation guarantees for the problem. To describe the algorithm, we will need some definitions.

Definition 3.2 (Star, Price, and Maximal Star) *A* star $\mathcal{S} = (i, C')$ *consists of a facility i and a subset $C' \subseteq C$. The* price *of $\mathcal{S}$ is* $\mathsf{price}(\mathcal{S}) = (f_i + \sum_{j \in C'} d(j, i))/|C'|$. *A star $\mathcal{S}$ is said to be* maximal *if all* strict *super sets of C' have a larger price, i.e., for all $C'' \supsetneq C'$,* $\mathsf{price}((i, C'')) > \mathsf{price}((i, C'))$.

The greedy algorithm of Jain et al. proceeds as follows:

> Until no client remains, pick the cheapest star (i, C'), open the facility i, set $f_i = 0$, remove all clients in C' from the instance, and repeat.

This algorithm has a sequential running time of $O(m \log m)$ and using techniques known as factor-revealing LP, Jain et al. show that the algorithm has an approximation factor of 1.861 [JMM+03]. From a parallelization point of view, the algorithm is highly sequential—at each step, only the minimum-cost option is chosen, and every subsequent step depends on the preceding one. In this section, we describe how to overcome this sequential nature and obtain an RNC algorithm inspired by the greedy algorithm of Jain et al. We show that the parallel algorithm is a $(3.722 + \varepsilon)$-approximation.

The key idea to parallelization is that much faster progress will be made if we allow a small slack in what can be selected in each round; however, a subselection step is necessary to ensure that facility and connection costs are properly accounted for.

Algorithm 3.3.1 Parallel greedy algorithm for metric facility location.

In rounds, the algorithm performs the following steps until no client remains:

```
1. For each facility $i$, **in parallel**, compute $\mathcal{S}_i = (i, C^{(i)})$, the lowest-priced maximal star
   centered at $i$.
2. Let $\tau = \min_{i \in F} \mathsf{price}(\mathcal{S}_i)$, and let $I = \{i \in F : \mathsf{price}(\mathcal{S}_i) \leq \tau(1+\varepsilon)\}$.
3. Construct a bipartite graph $H = (I, C', \{ij : d(i,j) \leq \tau(1+\varepsilon)\})$, where $C' = \{j \in
   C : \exists i \in I \text{ s.t. } d(i,j) \leq \tau(1+\varepsilon)\}$.
4. **Facility Subselection**: while $(I \neq \emptyset)$:
   (a) Let $\Pi : I \to \{1, \ldots, |I|\}$ be a random permutation of $I$.
   (b) For each $j \in C'$, let $\varphi_j = \arg\min_{i \in N_H(j)} \Pi(i)$.
   (c) For each $i \in I$, if $|\{j : \varphi_j = i\}| \geq \frac{1}{2(1+\varepsilon)} \deg(i)$, add $i$ to $F_A$ (open $i$), set
       $f_i = 0$, remove $i$ from $I$ , and remove $N_H(i)$ from both $C$ and $C'$.
       *Note: In the analysis, the clients removed in this step have $\pi_j$ set as follows. If
       the facility $\varphi_j$ is opened, let $\pi_j = \varphi_j$; otherwise, $\pi_j$ is set to any facility $i$ we
       open in this step such that $ij \in E(H)$. Note that any facility that is opened is
       at least $1/(2(1+\varepsilon))$ paid for by the clients that select it, and that since every
       client is assigned to at most one facility, they only pay for one edge.*
   (d) Remove $i \in I$ (and the incident edges) from the graph $H$ if on the remaining
       graph, $\frac{f_i + \sum_{j \in N_H(i)} d(j,i)}{\deg(i)} > \tau(1+\varepsilon)$. These facilities will show up in the next
       round (outer-loop).
   *Note: After $f_i$ is set to $0$, facility $i$ will still show up in the next round.*
```

We present the parallel algorithm in Algorithm 3.3.1 and now describe step 1 in greater detail; steps 2 – 3 can be implemented using standard techniques [JáJ92, Lei92]. As observed in Jain et al. [JMM$^+$03] (see also Fact 3.3), for each facility i, the lowest-priced star centered at i consists of the κ_i closest clients to i, for some κ_i. Following this observation, we can presort the distance between facilities and clients for each facility. Let i be a facility and assume without loss of generality that $d(i,1) \leq d(i,2) \leq \cdots \leq d(i,n_c)$. Then, the cheapest maximal star for this facility can be found as follows. Using prefix sum, compute the sequence $p^{(i)} = \{(f_i + \sum_{j \leq k} d(i,k))/k\}_{k=1}^{n_c}$. Then, find the smallest index k such that $p_k^{(i)} < p_{k+1}^{(i)}$ or use $k = n_c$ if no such index exists. It is easy to see that the maximal lowest-priced star centered at i is the facility i together with the client set $\{1, \ldots, k\}$.

Crucial to this algorithm is a subselection step, which ensures that every facility and the clients that connect to it are adequately accounted for in the dual-fitting analysis. This subselection process can be seen as scaling back on the aggressiveness of opening up the facilities, mimicking the greedy algorithm's behavior more closely.

3.3.1 Analysis

We present a dual-fitting analysis of the above algorithm. The analysis relies on the client-to-facility assignment π, defined in the description of the algorithm. The following easy-to-check facts will be useful in the analysis.

Fact 3.3 *For each iteration of the execution, the following holds: (1) If $\mathcal{S}_i$ is the cheapest maximal star centered at i, then j appears in $\mathcal{S}_i$ if and only if $d(j,i) \le \mathsf{price}(\mathcal{S}_i)$. (2) If $t = \mathsf{price}(\mathcal{S}_i)$, then $\sum_{j\in C} \max(0, t - d(j,i)) = f_i$.*

Now consider the dual program in Figure 3.1. For each client j, set α_j to be the τ setting in the iteration that the client was removed. We begin the analysis by relating the cost of the solution that the algorithm outputs to the cost of the dual program.

Lemma 3.4 *The cost of the algorithm's solution $\sum_{i\in F_A} f_i + \sum_{j\in C} d(j, F_A)$ is upper-bounded by $2(1+\varepsilon)^2 \sum_{j\in C} \alpha_j$.*

Proof: Consider that in step 4(c), a facility i is opened if at least a $\frac{1}{2(1+\varepsilon)}$ fraction of the neighbors "chose" i. Furthermore, we know from the definition of H that, in that round, $f_i + \sum_{j\in N_H(i)} d(j,i) \le \tau(1+\varepsilon)\deg(i)$. By noting that we can partition C by which facility the client is assigned to in the assignment π, we establish

$$\begin{aligned}\sum_{j\in C} \alpha_j \cdot 2(1+\varepsilon)^2 &\ge \sum_{i\in F_A} \Big(f_i + \sum_{j:\pi_j = i} d(j,i)\Big)\\ &\ge \sum_{i\in F_A} f_i + \sum_{j\in C} d(j, F_A),\end{aligned}$$

as desired. ∎

In the series of claims that follows, we show that when scaled down by a factor of $\gamma = 1.861$, the α setting determined above is a dual feasible solution. We will assume without loss of generality that $\alpha_1 \le \alpha_2 \le \cdots \le \alpha_{n_c}$. Let $W_i = \{j \in C : \alpha_j \ge \gamma \cdot d(j,i)\}$ for all $i \in F$ and $W = \cup_i W_i$.

Claim 3.5 *For any facility $i \in F$ and client $j_0 \in C$,*

$$\sum_{j \in W : j \geq j_0} \max(0, \alpha_{j_0} - d(j,i)) \quad \leq \quad f_i.$$

Proof: Suppose for a contradiction that there exist client j and facility i such that the inequality in the claim does not hold. That is,

$$\sum_{j \in W : j \geq j_0} \max(0, \alpha_{j_0} - d(j,i)) \quad > \quad f_i. \tag{3.3}$$

Consider the iteration in which τ is α_{j_0}; call this iteration ℓ. By Equation (3.3), there exists a client $j \in W \cap \{j \in \mathbb{Z}_+ : j \geq j_0\}$ such that $\alpha_{j_0} - d(j,i) > 0$; thus, this client participated in a star in an iteration prior to ℓ and was connected up. Therefore, it must be the case that $\alpha_j < \alpha_{j_0}$, which is a contradiction to our assumption that $j_0 \leq j$ and $\alpha_1 \leq \alpha_2 \leq \ldots . \alpha_{n_c}$. ■

Claim 3.6 *Let $i \in F$, and $j, j' \in W$ be clients. Then, $\alpha_j \leq \alpha_{j'} + d(i,j') + d(i,j)$.*

The proof of this claim closely parallels that of Jain et al. [JMM+03] and is omitted. These two claims form the basis for the set up of Jain et al.'s factor-revealing LP. Hence, combining them with Lemmas 3.4 and 3.6 of Jain et al. [JMM+03], we have the following lemma:

Lemma 3.7 *The setting $\alpha'_j = \frac{\alpha_j}{\gamma}$ and $\beta'_{ij} = \max(0, \alpha'_j - d(j,i))$ is a dual feasible solution, where $\gamma = 1.861$.*

An Alternative Proof Without Factor-Revealing LP. We note that a slightly weaker result can be derived without the use of factor-revealing LP. Claims 3.6 and 3.5 can be combined to prove the following lemma:

Lemma 3.8 *The setting $\alpha'_j = \alpha_j/3$ and $\beta'_{ij} = \max(0, \alpha'_j - d(j,i))$ is a dual feasible solution.*

Proof: We will show that for each facility $i \in F$,

$$\sum_{j \in W_i} \left(\alpha_j - 3 \cdot d(j,i)\right) \quad \leq \quad 3 \cdot f_i. \tag{3.4}$$

Note that if W_i is empty, the lemma is trivially true. Thus, we assume W_i is non-empty and define j_0 to be $\min W_i$. Since $j_0 \in W_i$, $d(j_0, i) \leq \alpha_{j_0}$ by the definition of W_i. Now let $T = \{j \in W_i : \alpha_{j_0} \geq d(j,i)\}$. Applying Claims 3.6 and 3.5, we have

$$\begin{aligned}\sum_{j\in W_i}(\alpha_j - d(j,i)) \leq \sum_{j\in W_i}(\alpha_{j_0} + d(j_0,i)) &\leq \sum_{j\in W_i} 2\cdot\alpha_{j_0} \\ &\leq 2f_i + \sum_{j\in T} 2\cdot d(j,i) + \sum_{j\in W_i\setminus T} 2\cdot d(j,i) \leq 2f_i + \sum_{j\in W_i} 2\cdot d(j,i),\end{aligned}$$

which proves inequality (3.4). With this, it is easy to see that our choice of β'_{ij}'s ensures that all constraints of the form $\alpha_j - \beta_{ij} \leq d(j,i)$ are satisfied. Then, by inequality (3.4), we have $\sum_{j\in C}\max(0, \alpha_j - 3\cdot d(j,i)) = \sum_{j\in W_i}[\alpha_j - 3\cdot d(j,i)] \leq 3\cdot f_i$, which implies that $\sum_{j\in C}\max(0,\alpha_j - 3\cdot d(j,i)) \leq 3\cdot f_i$. Hence, we conclude that for all facility $i \in F$, $\sum_{j\in C}\beta'_{ij} \leq f_i$, proving the lemma. ■

Running Time Analysis

Consider the algorithm's description in Algorithm 3.3.1. The rows can be presorted to give each client its distances from facilities in order. In the original order, each element can be marked with its rank. Step 1 then involves a prefix sum on the sorted order to determine how far down the order to go and then selection of all facilities at or below that rank. Steps 2–3 require reductions and distributions across the rows or columns of the matrix. The subset $I \subset F$ can be represented as a bit mask over F. Step 4 is more interesting to analyze; the following lemma bounds the number of rounds facility subselection is executed, the proof of which is analogous to Lemma 4.1.2 of Rajagopalan and Vazirani [RV98]; we present here for completeness a simplified version of their proof, which suffices for our lemma.

Lemma 3.9 *With probability $1 - o(1)$, the subselection step terminates within $O(\log_{1+\varepsilon} m)$ rounds.*

Proof: Let $\Phi = |E|$. We will show that if Φ' is the potential value after an iteration of the subselection step, then $\mathbf{E}[\Phi - \Phi'] \geq c\Phi$, for some constant $c > 0$. The lemma then follows from standard results in probability theory. To proceed, define $\text{chosen}_i = |\{j \in C' : \varphi_j = i\}|$. Furthermore, we say that an edge ij is *good* if at most $\theta = \frac{1}{2}(1 - \frac{1}{1+\varepsilon})$ fraction of neighbors of i have degree higher than j.

Consider a good edge ij. We will estimate $\mathbf{E}\left[\mathsf{chosen}_i | \varphi_j = i\right]$. Since ij is good, we know that

$$\sum_{j' \in N_H(i)} \mathbf{1}_{\{\deg(j') \leq \deg(j)\}} \geq (1-\theta)\deg(i).$$

Therefore, $\mathbf{E}\left[\mathsf{chosen}_i | \varphi_j = i\right] \geq \frac{1}{2}(1-\theta)\deg(i)$, as it can be shown that $\mathbf{Pr}\left[\varphi_{j'} = i | \varphi_j = i\right] \geq \frac{1}{2}$ for all $j' \in N_H(i)$ and $\deg(j') \leq \deg(j)$. By Markov's inequality and realizing that $\mathsf{chosen}_i \leq \deg(i)$, we have

$$\mathbf{Pr}\left[\mathsf{chosen}_i \geq \frac{1}{2(1+\varepsilon)}\deg(i) \;\middle|\; \varphi_j = i\right] = p_0 > 0.$$

Finally, we note that $\mathbf{E}\left[\Phi - \Phi'\right]$ is at least

$$\begin{aligned}
&\sum_{ij \in E} \mathbf{Pr}\left[\varphi_j = i \text{ and } \mathsf{chosen}_i \geq \frac{1}{2(1+\varepsilon)}\deg(i)\right] \cdot \deg(j) \\
&\geq \sum_{\text{good } ij \in E} \frac{1}{\deg(j)} p_0 \deg(j) \\
&\geq p_0 \sum_{ij \in E} \mathbf{1}_{\{ij \text{ is good}\}}.
\end{aligned}$$

Since at least θ fraction of the edges are good, $\mathbf{E}\left[\Phi - \Phi'\right] \geq p_0 \theta \Phi$. Since $\ln(1/(1-p_0\theta)) = \Omega(\log(1+\varepsilon))$, the lemma follows from standard results in probability [MR95]. ■

It is easy to see that each subselection step can be performed with a constant number of basic matrix operations over the D matrix. Therefore, if the number of rounds the main body is executed is r, the algorithm makes $O(r \log_{1+\varepsilon} m)$ calls to the basic matrix operations described in Section 3.1 with probability exceeding $1 - o(1)$. It also requires a single sort in the preprocessing. This means a total of $O(rm \log_{1+\varepsilon} m)$ work (with probability exceeding $1-o(1)$) on the EREW PRAM. Furthermore, it is cache efficient (cache complexity is $O(w/B)$) since the sort is only applied once and does not dominate the cache bounds.

Bounding the number of rounds

Before describing a less restrictive alternative, we point out that the simplest way to bound the number of rounds by a polylogarithm factor is to rely on the common assumption that the facility cost, as well as the ratio between the minimum (non-zero) and the maximum client-facility distance, is polynomially bounded in the input size. As a result of this assumption, the number of rounds is upper-bounded by $\log_{1+\varepsilon}(m^c) = O(\log_{1+\varepsilon} m)$, for some $c \geq 1$.

Alternatively, we can apply a preprocessing step to ensure that the number of rounds is polylogarithm in m. The basic idea of the preprocessing step is that if a star is "relatively cheap," opening it right away will harm the approximation factor only slightly. Using the bounds in Equation (3.2), if $\mathcal{S}_i$ is the lowest-priced maximal star centered at i, we know we can afford to open i and discard all clients attached to it if $\mathsf{price}(\mathcal{S}_i) \leq \frac{\gamma}{m^2}$. Therefore, the preprocessing step involves: (1) computing $\mathcal{S}_i$, the lowest-priced maximal star centered at i, for all $i \in F$, (2) opening all i such that $\mathsf{price}(\mathcal{S}_i) \leq \frac{\gamma}{m^2}$, (3) setting f_i of these facilities to 0 and removing all clients attached to these facilities.

Computing γ takes $O(\log n_c + \log n_f)$ depth and $O(m)$ work. The rest of the preprocessing step is at most as costly as a step in the main body. Thus, the whole preprocessing step can be accomplished in $O(\log m)$ depth and $O(m)$ work. With this preprocessing step, three things are clear: First, τ in the first iteration of the main algorithm will be at least $\frac{\gamma}{m^2}$, because cheaper stars have already been processed in preprocessing. Second, the cost of our final solution is increased by at most $n_c \times \frac{\gamma}{m^2} \leq \frac{\gamma}{m} \leq \mathsf{opt}/m$, because the facilities and clients handled in preprocessing can be accounted for by the cost of their corresponding stars—specifically, there can be most n_c stars handled in preprocessing, each of which has price $\leq \gamma/m^2$; and the price for a star includes both the facility cost and the connection cost of the relevant clients and facilities. Finally, in the final iteration, $\tau \leq n_c\gamma$. As a direct consequence of these observations, the number of rounds is upper-bounded by $\log_{1+\varepsilon}(\frac{n_c\gamma}{\gamma/m^2}) \leq \log_{1+\varepsilon}(m^3) = O(\log_{1+\varepsilon} m)$, culminating in the following theorem:

Theorem 3.10 *Let $0 < \varepsilon \leq 1$ be fixed. For sufficiently large input, there is a greedy-style* RNC *$O(m \log^2_{1+\varepsilon}(m))$-work algorithm that yields a factor-$(3.722 + \varepsilon)$ approximation for the metric facility-location problem.*

3.4 Facility Location: Primal-Dual

The primal-dual scheme is a versatile paradigm for combinatorial algorithms design. In the context of facility location, this scheme underlies the Lagrangian-multiplier preserving[1] (LMP) 3-approximation algorithm of Jain and Vazirani, enabling them to use the algorithm as a subroutine in their 6-approximation algorithm for k-median [JV01b].

The algorithm of Jain and Vazirani consists of two phases, a primal-dual phase and a post-processing phase. To summarize this algorithm, consider the primal and dual programs in Figure 3.1. In the primal-dual phase, starting with all dual variables set to 0, we raise the

[1]This means $\alpha \sum_{i \in F_A} f_i + \sum_{j \in C} d(j, F_A) \leq \alpha \cdot \mathsf{opt}$, where α is the approximation ratio.

dual variables α_j's uniformly until a constraint of the form $\alpha_j - \beta_{ij} \leq d(j,i)$ becomes tight, at which point β_{ij} will also be raised, again, uniformly to prevent these constraints from becoming overtight. When a constraint $\sum_j \beta_{ij} \leq f_i$ is tight, facility i is tentatively opened and clients with $\alpha_j \geq d(j,i)$ are "frozen," i.e., we stop raising their α_j values from this point on. The first phase ends when all clients are frozen. In the postprocessing phase, we compute and output a maximal independent set on a graph G of tentatively open facilities; in this graph, there is an edge between a pair of facilities i and i' if there is a client j such that $\alpha_j > d(j,i)$ and $\alpha_j > d(j,i')$. Thus, the maximal independent set ensures proper accounting of the facility cost (i.e., each client "contributes" to at most one open facility, and every open facility has enough contribution). Informally, we say that a client j "pays" for or "contributes" to a facility i if $\beta_{ij} = \alpha_j - d(j,i) > 0$.

Remarks. We note that in the parallel setting, the description of the postprocessing step above does not directly lead to an efficient algorithm, because constructing G in polylogarithmic depth seems to need $O(mn_f)$ work, which is much more than one needs sequentially.

In this section, we show how to obtain a work-efficient RNC $(3+\varepsilon)$-approximation algorithm for facility location, based on the primal-dual algorithm of Jain and Vazirani. Critical to bounding the number of iterations in the main algorithm by $O(\log m)$ is a preprocessing step, which is similar to that used by Pandit and Pemmaraju in their distributed algorithm [PP09].

Preprocessing: Assuming γ as defined in Equation (3.2), we will open every facility i that satisfies

$$\sum_{j \in C} \max\left(0, \frac{\gamma}{m^2} - d(j,i)\right) \quad \geq \quad f_i.$$

Furthermore, for all clients j such that there exists an opened i and $d(j,i) \leq \gamma/m^2$, we declare them connected and set $\alpha_j = 0$. The facilities opened in this step will be called *free* facilities and denoted by the set F_0.

Main Algorithm: The main body of the algorithm is described in Algorithm 3.4.1. The algorithm outputs a bipartite graph $H = (F_T, C, E)$, constructed as the algorithm executes. Here F_T is the set of facilities declared open during the iterations of the main algorithm and E is given by $E = \{ij : i \in F, j \in C, \text{ and } (1+\varepsilon)\alpha_j > d(j,i)\}$.

Post-processing. As a post-processing step, we compute $I = \textsc{MaxUDom}(H)$. Thus, the set of facilities $I \subseteq F_T$ has the property that each client contributes to the cost of at most one facility in I. Finally, we report $F_A = I \cup F_0$ as the set of facilities in the final solution.

Algorithm 3.4.1 Parallel primal-dual algorithm for metric facility location

For iteration $\ell = 0, 1, \dots,$ the algorithm performs the following steps until all facilities are opened or all clients are frozen, whichever happens first.

1. For each unfrozen client j, **in parallel**, set α_j to $\frac{\gamma}{m^2}(1+\varepsilon)^\ell$.
2. For each unopened facility i, **in parallel**, declare it open if

$$\sum_{j \in C} \max(0, (1+\varepsilon)\alpha_j - d(j,i)) \geq f_i.$$

3. For each unfrozen client j, **in parallel**, freeze this client if there exists an opened facility i such that $(1+\varepsilon)\alpha_j \geq d(j,i)$.
4. Update the graph H by adding edges between pairs of nodes ij such that $(1+\varepsilon)\alpha_j > d(j,i)$.

After the last iteration, if all facilities are opened but some clients are *not* yet frozen, we determine in parallel the α_j settings of these clients that will make them reach an open facility (i.e., $\alpha_j = \min_i d(j,i)$). Finally, update the graph H as necessary.

3.4.1 Analysis

To analyze approximation guarantee of this algorithm, we start by establishing that the α_j setting produced by the algorithm leads to a dual feasible solution.

Claim 3.11 *For any facility i,*

$$\sum_{j \in N_H(i)} \max(0, \alpha_j - d(j,i)) \leq f_i.$$

Proof: Let $\alpha_j^{(\ell)}$ denote the α_j value at the end of iteration ℓ. Suppose for a contradiction that there is a facility i which is overtight. More formally, there exists $i \in F$ and the smallest ℓ such that $\sum_{j \in N_F(i)} \max(0, \alpha_j^{(\ell)} - d(j,i)) > f_i$. Let J be the set of unfrozen neighboring clients of i in iteration $\ell - 1$. The reason facility i was not opened in iteration $\ell - 1$ and the surrounding clients were not frozen is

$$\textit{raised}_i \overset{\text{def}}{=} \sum_{j \in N_F(i) \setminus J} \max(0, (1+\varepsilon)\alpha_j^{(\ell-1)} - d(j,i)) + \sum_{j \in J} \max(0, (1+\varepsilon)t_{\ell-1} - d(j,i)) < f_i.$$

However, we know that $t_\ell = (1+\varepsilon)t_{\ell-1}$, and for each frozen neighboring client j (i.e., $j \in N_F(i) \setminus J$), $\alpha_j^{(\ell)} = \alpha_j^{(\ell-1)}$, so

$$\textit{raised}_i \geq \sum_{j \in N(i) \setminus J} \max(0, \alpha_j^{(\ell)} - d(j,i)) + \sum_{j \in J} \max(0, t_\ell - d(j,i)) = \sum_{j \in N(i)} \max(0, \alpha_j^{(\ell)} - d(j,i)),$$

which is a contradiction. ■

It follows from this claim that setting $\beta_{ij} = \max(0, \alpha_j - d(j,i))$ provides a dual feasible solution. Next we relate the cost of our solution to the cost of the dual solution. To ease the following analyses, we use a client-to-facility assignment $\pi : C \to F$, defined as follows: For all $j \in C$, let $\varphi(j) = \{i : (1+\varepsilon)\alpha_j \geq d(j,i)\}$. Now for each client j, **(1)** if there exists $i \in F_0$ such that $d(j,i) \leq \gamma/m^2$, set π_j to *any* such i; **(2)** if there exists $i \in I$ such that ij is an edge in H, then $\pi_j = i$ (i is unique because of properties of I) ; **(3)** if there exists $i \in I$ such that $i \in \varphi(j)$, then $\pi_j = i$; **(4)** otherwise, pick $i' \in \varphi(j)$ and set π_j to $i \in I$ which is a neighbor of a neighbor of i'.

Clients of the first case, denoted by C_0, are called *freely connected*; clients of the cases (2) and (3), denoted by C_1, are called *directly connected.* Otherwise, a client is *indirectly connected.*

The following lemmas bound the facility costs and the connection costs of indirectly connected clients.

Lemma 3.12

$$\sum_{i \in F_A} f_i \;\leq\; \frac{\gamma}{m} + \sum_{j \in C_1} (1+\varepsilon)\alpha_j - \sum_{j \in C_0 \cup C_1} d(j, \pi_j)$$

Proof: When facility $i \in F_T$ was opened, it must satisfy $f_i \leq \sum_{j : ij \in E(G)} (1+\varepsilon)\alpha_j - d(j,i)$. If client j has contributed to i (i.e., $(1+\varepsilon)\alpha_j - d(j,i) > 0$) and $i \in I$, then j is directly connected to it. Furthermore, for each client j, there is at most one facility in I that it contributes to (because $I = \textsc{MaxUDom}(H)$). Therefore, $\sum_{i \in I} f_i \leq \sum_{j \in C_1} (1+\varepsilon)\alpha_j - d(j, \pi_j)$. Furthermore, for each "free" facility, we know that $f_i \leq \sum_{j \in C} \max(0, \gamma^2/m^2 - d(j,i))$, so by our choice of π, $f_i \leq \frac{\gamma}{m^2} \times n_c - \sum_{j \in C_0 : \pi_j = i} d(j,i)$. Thus, $\sum_{i \in F_0} f_i \leq \gamma/m - \sum_{j \in C_0} d(j,i)$. Combining these results and observing that F_A is the disjoint union of I and F_0, we have the lemma. ■

Lemma 3.13 *For each indirectly connected client j (i.e., $j \notin C_0 \cup C_1$), we have $d(j, \pi_j) \leq 3(1+\varepsilon)\alpha_j$.*

Proof: Because $j \notin C_0 \cup C_1$ and $I = \textsc{MaxUDom}(H)$, there must exist a facility $i' \in \varphi(j)$ and a client j' such that j' contributed to both i and i', and $(1+\varepsilon)\alpha_j \geq d(j, i')$. We claim that both $d(j', i')$ and $d(j', i)$ are upper-bounded by $(1+\varepsilon)\alpha_j$. To see this, we note that because j' contributed to both i and i', $d(j', i') \leq (1+\varepsilon)\alpha_{j'}$ and $d(j', i) \leq (1+\varepsilon)\alpha_{j'}$.

Let ℓ be the iteration in which j was declared frozen, so $\alpha_j = t_\ell$. Since $i' \in \varphi(j)$, i' must be declared open in iteration $\leq \ell$. Furthermore, because $(1+\varepsilon)\alpha_{j'} > d(j', i')$, $\alpha_{j'}$ must be frozen in or prior to iteration ℓ. Consequently, we have $\alpha_{j'} \leq t_\ell = \alpha_j$. Combining these facts and applying the triangle inequality, we get $d(j,i) \leq d(j,i') + d(i',j') + d(j',i) \leq (1+\varepsilon)\alpha_j + 2(1+\varepsilon)\alpha_{j'} \leq 3(1+\varepsilon)\alpha_j$. ■

By Lemmas 3.12 and 3.13, we establish

$$3\sum_{i\in F_A} f_i + \sum_{j\in C} d(j,\pi_j) \quad\leq\quad \frac{3\gamma}{m} + 3(1+\varepsilon)\sum_{j\in C}\alpha_j. \tag{3.5}$$

Now since $\{\alpha_j, \beta_{ij}\}$ is dual feasible, its value can be at most that of the primal optimal solution; that is, $\sum_j \alpha_j \leq \text{opt}$. Therefore, combining with Equation (3.5), we know that the cost of the solution returned by parallel primal-dual algorithm in this section is at most $3\sum_{i\in F_A} f_i + \sum_{j\in C} d(j,C) \leq (3+\varepsilon')\text{opt}$ for some $\varepsilon' > 0$ when the problem instance is large enough.

Running Time Analysis

We analyze the running of the algorithm presented, starting with the main body of the algorithm. Since $\sum_j \alpha_j \leq \text{opt}$ and $\text{opt} \leq n_c\gamma$, no α_j can be bigger than $n_c\gamma \leq m\gamma$. Hence, the main algorithm must terminate before $\ell > 3\log_{1+\varepsilon} m$, which upper-bounds the number of iterations to $O(\log_{1+\varepsilon} m)$. In each iteration, steps 1, 3, and 4 perform trivial work. Step 2 can be broken down into (1) computing the max for all $i \in F, j \in C$,, and (2) computing the sum for each $i \in F$. These can all be implemented with the basic matrix operations, giving a totalof $O(\log_{1+\varepsilon} m)$ of basic matrix operations over a matrix of size m.

The preprocessing step, again, involves some reductions over the rows and columns of the matrix. This includes the calculations of γ_j's and the composite γ. The post-processing step relies on computing the U-dominating set, as described in Section 3.1 which runs in $O(\log m)$ matrix operations.

The whole algorithm therefore runs in $O(\log_{1+\varepsilon} m)$ basic matrix operations and is hence work efficient compared to the $O(m \log m)$ sequential algorithm of Jain and Vazirani. Putting these altogether, we have the following theorem:

Theorem 3.14 *Let $\varepsilon > 0$ be fixed. For sufficiently large m, there is a primal-dual* RNC *$O(m\log_{1+\varepsilon} m)$-work algorithm that yields a factor-$(3+\varepsilon)$ approximation for the metric facility-location problem.*

3.5 Other Results

In this section, we consider other applications of dominator set in facility-location problems.

3.5.1 k-Center

Hochbaum and Shmoys [HS85] show a simple factor-2 approximation for k-center. The algorithm performs a binary search on the range of distances. We show how to combine the dominator-set algorithm from Section 3.2 with standard techniques to parallelize the algorithm of Hochbaum and Shmoys, resulting in an RNC algorithm with the same approximation guarantee. Consider the set of distances $\mathcal{D} = \{d(i,j) : i \in C \text{ and } j \in V\}$ and order them so that $d_1 < d_2 < \cdots < d_p$ and $\{d_1, \ldots, d_p\} = \mathcal{D}$, where $p = |\mathcal{D}|$. The sequence $\{d_i\}_{i=1}^{p}$ can be computed in $O(\log |V|)$ depth and $O(|V|^2 \log |V|)$ work. Let H_α be a graph defined as follows: the nodes of H_α is the set of nodes V, but there is an edge connecting i and j if and only if $d(i,j) \leq \alpha$.

The main idea of the algorithm is simple: find the smallest index $t \in \{1, 2, \ldots, p\}$ such that $\textsc{MaxDom}(H_{d_t}) \leq k$. Hochbaum and Shmoys observe that the value t can be found using binary search in $O(\log p) = O(\log |V|)$ probes. We parallelize the probe step, consisting of constructing $H_{d_{t'}}$ for a given $t' \in \{1, \ldots, p\}$ and checking whether $|\textsc{MaxDom}(H_{d_{t'}})|$ is bigger than k. Constructing $H_{d_{t'}}$ takes $O(1)$ depth and $O(|V|^2)$ work, and using the maximal-dominator-set algorithm from Section 3.2, the test can be performed in expected $O(\log^2 |V|)$ depth and expected $O(|V|^2 \log |V|)$ work. The approximation factor is identical to the original algorithm, hence proving the following theorem:

Theorem 3.15 *There is an* RNC 2*-approximation algorithm with* $O((|V| \log |V|)^2)$ *work for* k*-center.*

3.5.2 Facility Location: LP Rounding

LP rounding was among the very first techniques that yield non-trivial approximation guarantees for metric facility location. The first constant-approximation algorithm was given by Shmoys et al. [STA97]. Although we do not know how to solve the linear program for facility location in polylogarithmic depth, we demonstrate another application of the dominator-set algorithm and the slack idea by parallelizing the randomized-rounding step of Shmoys et al. The algorithm yields a $(4 + \varepsilon)$-approximation, and the randomized rounding is an RNC algorithm.

The randomized rounding algorithm of Shmoys et al. consists of two phases: a filtering phase and a rounding phase. In the following, we show how to parallelize these phases and prove that the parallel version has a similar guarantee. Our presentation differs slightly from the original work but works in the same spirit.

Filtering: The filtering phase is naturally parallelizable. Fix α to be a value between 0 and 1. Given an optimal primal solution (x, y), the goal of this step is to produce a new solution (x', y') with properties as detailed in Lemma 3.16. Let $\delta_j = \sum_{i \in F} d(i,j) \cdot x_{ij}$, $B_j = \{i \in F : d(i,j) \leq (1+\alpha)\delta_j\}$, and $\mathsf{mass}(B_j) = \sum_{i \in B_j} x_{ij}$. We compute x'_{ij} and y'_i as follows: (1) let $x'_{ij} = x_{ij}/\mathsf{mass}(B_j)$ if $i \in B_j$ or 0 otherwise, and (2) let $y'_i = \min(1, (1+1/\alpha)y_i)$.

Lemma 3.16 *Given an optimal primal solution (x, y), there is a primal feasible solution (x', y') such that (1) $\sum_i x'_{ij} = 1$, (2) if $x'_{ij} > 0$, then $d(j,i) \leq (1+\alpha)\delta_j$, and (3) $\sum_i f_i y_i \leq (1 + \frac{1}{\alpha}) \sum_i f_i y'_i$.*

Proof: By construction, (1) clearly holds. Furthermore, we know that if $x'_{ij} > 0$, it must be the case that $i \in B_j$, so $d(j,i) \leq (1+\alpha)\delta_j$, proving (2). By definition of y'_i, $\sum_i f_i y_i \leq (1 + \frac{1}{\alpha}) \sum_i f_i y'_i$, proving (3). Finally, since in an optimal LP solution, $\sum_i x_{ij} = 1$, we know that $\mathsf{mass}(B_j) \geq \frac{\alpha}{1+\alpha}$, by an averaging argument. Therefore, $x'_{ij} \leq (1 + \frac{1}{\alpha}) x_{ij} \leq \min(1, (1+\frac{1}{\alpha})y_i) = y'_i$, showing that (x', y') is primal feasible. ∎

Rounding: The rounding phase is more challenging to parallelize because it is inherently sequential—a greedy algorithm which considers the clients in an increasing order of δ_j and appears to need $\Omega(n_c)$ steps. We show, however, that we can achieve parallelism by eagerly processing the clients $S = \{j : \delta_j \leq (1+\varepsilon)\tau\}$. This is followed by a clean-up step, which uses the dominator-set algorithm to rectify the excess facilities. We precompute the following information: (1) for each j, let i_j be the least costly facility in B_j, and (2) construct $H = (C, F, ij \in E \text{ iff. } i \in B_j)$.

There is a preprocessing step to ensure that the number of rounds is polylogarithmic in m. Let θ be the value of the optimal LP solution. By an argument similar to that of Section 3.3, we can afford to process all clients with $\delta_j \leq \theta/m^2$ in the first round, increasing the final cost by at most $\theta/m \leq \mathsf{opt}/m$. The algorithm then proceeds in rounds, each performing the following steps:

Since J is U-dominator of H, we know that for all distinct $j, j' \in J$, $B_j \cap B_{j'} = \emptyset$; therefore, $\sum_{i \in I} f_i = \sum_{j \in J} f_{i_j} \leq \sum_{j \in J} \left(\sum_{i \in B_j} x'_{ij} f_{i_j}\right) \leq \sum_{j \in J} y'_i f_{i_j} \leq \sum_{j \in J} y'_i f_i$, proving the following claim:

1. Let $\tau = \min_j \delta_j$.
2. Let $S = \{j : \delta_j \leq (1+\varepsilon)\tau\}$ and
3. Let $J = \text{MaxUDom}(H)$, add $I = \{i_j : j \in J\}$ to F_A; finally, remove all of S and $\cup_{j \in S} B_j$ from $V(H)$.

Claim 3.17 *In each round,* $\sum_{i \in I} f_i \leq \sum_{i \in \cup_j B_j} y_i' f_i$.

Like our previous analyses, we will define a client-to-facility assignment π convenient for the proof. For each $j \in C$, if $i_j \in F_A$, let $\pi_j = i_j$; otherwise, set $\pi_j = i_{j'}$, where j' is the client that causes i_j to be shut down (i.e., either $i_j \in B_{j'}$ and j' was process in a previous iteration, or both j and j' are processed in the same iteration but there exists $i \in B_j \cap B_{j'}$).

Claim 3.18 *Let j be a client. If $i_j \in F_A$, then $d(j, \pi_j) \leq (1+\alpha)\delta_j$; otherwise, $d(j, \pi_j) \leq 3(1+\alpha)(1+\varepsilon)\delta_j$.*

Proof: If $i_j \in F_A$, then by Lemma 3.16, $d(j, \pi_j) \leq (1+\alpha)\delta_j$. If $i_j \notin F_A$, we know that there must exist $i \in B_j$ and j' such that $i \in B_{j'}$ and $\delta_{j'} \leq (1+\varepsilon)\delta_j$. Thus, applying Lemma 3.16 and the triangle inequality, we have $d(j, \pi_j) \leq d(j, i) + d(i, j') + d(j', i_{j'}) \leq 3(1+\alpha)(1+\varepsilon)\delta_j$. ■

Running Time Analysis: The above algorithm will terminate in at most $O(\log_{1+\varepsilon} m)$ rounds because the preprocessing step ensures the ratio between the maximum and the minimum δ_j values are polynomially bounded. Like previous analyses, steps 1 – 2 can be accomplished in O(1) basic matrix operations, and step 3 in $O(\log m)$ basic matrix operations on matrices of size m. This yields a total of $O(\log_{1+\varepsilon} m \log m)$ basic matrix operations, proving the following theorem:

Theorem 3.19 *Given an optimal LP solution for the primal LP in Figure 3.1, there is an* RNC *rounding algorithm yielding a $(4+\varepsilon)$-approximation with $O(m \log m \log_{1+\varepsilon} m)$ work. It is cache efficient.*

3.6 k-Median: Local Search

Local search, LP rounding, and Lagrangian relaxation are among the main techniques for approximation algorithms for k-median. In this section, building on the algorithms from previous sections, we present an algorithm for the k-median problem, based on local-search techniques. The natural local-search algorithm for k-median is very simple: starting with any set F_A of k facilities, find some $i \in F_A$ and $i' \in F \setminus F_A$ such that swapping them decreases the k-median cost, and repeat until no such moves can be found. Finding an improving swap or identifying that none exists takes $O(k(n-k)n)$ time sequentially, where n is the number of nodes in the instance. This algorithm is known to be a 5-approximation [AGK+04, GT08].

The key ideas in this section are that we can find a good initial solution S_0 quickly and perform each local-search step fast. Together, this means that only a small number of local-search steps is needed, and each step can be performed fast. To find a good initial solution, we observe that any optimal k-center solution is an n-approximation for k-median. Therefore, we will use the 2-approximation from Section 3.5.1 as a factor-$(2n)$ solution for the k-median problem. At the beginning of the algorithm, for each $j \in V$, we order the facilities by their distance from j, taking $O(n^2 \log n)$ work and $O(\log n)$ depth.

Let $0 < \varepsilon < 1$ be fixed. We say that a swap (i, i') such that $i \in F_A$ and $i' \in F \setminus F_A$ is *improving* if $\textsc{kMed}(F_S - i + i') < (1 - \beta/k)\textsc{kMed}(F_S)$, where $\beta = \varepsilon/(1+\varepsilon)$. The parallel algorithm proceeds as follows. In each round, find and apply an improving swap as long as there is one. We now describe how to perform each local-search step fast. During the execution, the algorithm keeps track of φ_j, the facility client j is assigned to, for all $j \in V$. We will consider all possible test swaps $i \in F_A$ and $i' \in V \setminus F_A$ *simultaneously in parallel*. For each potential swap (i, i'), every client can independently compute $\Delta_j = d(j, F_A - i + i') - d(j, F_A)$; this computation trivially takes $O(n_c)$ work and $O(1)$ depth, since we know φ_j and the distances are presorted. From here, we know that $\textsc{kMed}(F_A - i + i') - \textsc{kMed}(F_A) = \sum_j \Delta_j$, which can be computed in $O(n)$ work and $O(\log n)$ depth. Therefore, in $O(k(n-k)n)$ work and $O(\log n)$ depth, we can find an improving swap or detect that none exists. Finally, a round concludes by applying an improving swap to F_A and updating the φ_j values.

Arya et al. [AGK+04] show that the number of rounds is bounded by

$$O\left(\log_{1/(1-\beta/k)}\left(\textsc{kMed}(S_0)/\mathrm{opt}\right)\right) = O\left(\log_{1/(1-\beta/k)}(n)\right)$$

Since for $0 < \varepsilon < 1$, $\ln\left(1/(1-\beta/k)\right) \le \frac{2}{k}\ln\left(1/(1-\beta)\right)$, we have the following theorem, assuming $k \in O(\mathrm{polylog}(n))$, which is often the case in many applications:

Theorem 3.20 *For $k \in O(\mathrm{polylog}(n))$, there is an* NC $O(k^2(n-k)n\log_{1+\varepsilon}(n))$-work algorithm which gives a factor-$(5+\varepsilon)$ approximation for k-median.

Remarks. Relative to the sequential algorithm, this algorithm is work efficient—regardless of the range of k. In addition to k-median, this approach is applicable to k-means, yielding an $(81 + \varepsilon)$-approximation [GT08] in general metric spaces and a $(25 + \varepsilon)$-approximation for the Euclidean space [KMN+04], and the same parallelization techniques can be used to achieve the same running time. Furthermore, there is a factor-3 approximation local-search algorithm for facility location, in which a similar idea can be used to perform each local-search step efficiently; however, we do not know how to bound the number of rounds.

For completeness, we reproduce the proof of Gupta and Tangwongsan [GT08] that the natural local search algorithm yields a $(5 + \varepsilon)$-approximation. This proof can be generalized to k-means as well as higher ℓ_p-norms [GT08]

A Simple Analysis of k-Median Local Search

A Set of Test Swaps. To show that a local optimum is a good approximation, the standard approach is to consider a carefully chosen subset of potential swaps: if we are at a local optimum, each of these swaps must be non-improving. This gives us some information about the cost of the local optimum. To this end, consider the set F^* of facilities chosen by an optimum solution, and let F be the facilities at the local optimum. Without loss of generality, assume that $|F| = |F^*| = k$.

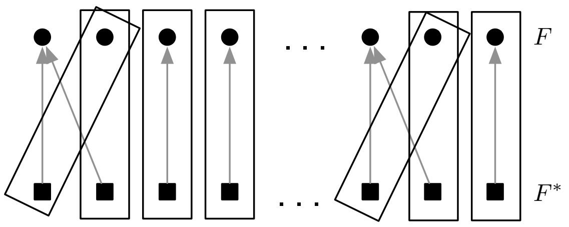


Figure 3.2: An example mapping $\eta\colon F^* \to F$ and a set of test swaps S.

Define a map $\eta : F^* \to F$ that maps each optimal facility f^* to a closest facility $\eta(f^*) \in F$: that is, $d(f^*, \eta(f^*)) \leq d(f^*, f)$ for all $f \in F$. Now define $R \subseteq F$ to be all the facilities that have *at most* 1 facility in F^* mapped to it by the map η. (In other words, if we create a directed bipartite graph by drawing an arc from f^* to $\eta(f^*)$, $R \subseteq F$ are those facilities whose in-degree is at most 1).

Finally, we define a set of k pairs $S = \{(r, f^*)\} \subseteq R \times F^*$ such that

- Each $f^* \in F^*$ appears in exactly one pair (r, f^*).
- If $\eta^{-1}(r) = \{f^*\}$ then r appears only once in S as the tuple (r, f^*).
- If $\eta^{-1}(r) = \emptyset$ then r appears at most in two tuples in S.

The procedure is simple: for each $r \in R$ with in-degree 1, construct the pair $(f, \eta^{-1}(r))$—let the optimal facilities that are already matched off be denoted by F_1^*. The other facilities in R have in-degree 0: denote them by R_0. A simple averaging argument shows that the unmatched optimal facilities $|F^* \setminus F_1^*| \leq 2|R_0|$. Now, arbitrarily create pairs by matching each node in R_0 to at most two pairs in $F^* \setminus F_1^*$ so that the above conditions are satisfied.

The following fact is immediate from the construction:

Fact 3.21 *For any tuple $(r, f^*) \in S$ and $\widehat{f^*} \in F$ with $\widehat{f^*} \neq f^*$, $\eta(\widehat{f^*}) \neq r$.*

Intuition for the Pairing.

To get some intuition for why the pairing S was chosen, consider the case when each facility in F is the closest to a unique facility in F^*, and far away from all other facilities in F^*—in this case, opening facility $f^* \in F^*$ and closing the matched facility in $f \in F$ can be handled by letting all clients attached to f be handled by f^* (or by other facilities in F). A problem case would be when a facility $f \in F$ is the closest to several facilities in F^*, since closing f and opening only one of these facilities in F^* might still cause us to pay too much—hence we never consider the gains due to closing such “popular” facilities, and instead only consider the swaps that involve facilities from the set of relatively “unpopular” facilities R.

Bounding the Cost of a Local Optimum

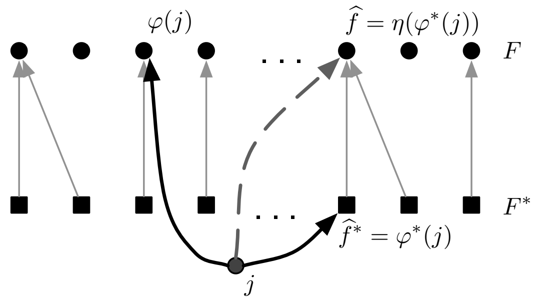
We use the fact that each of the swaps S are non-improving to show that that the local optimum has small cost.

Breaking ties arbitrarily, assume that $\varphi : V \to F$ and $\varphi^* : V \to F^*$ are functions mapping each client to some closest facility. For any client j, let $O_j = d(j, F^*) = d(j, \varphi^*(j))$ be the client j's cost in the optimal solution, and $A_j = d(j, F) = d(j, \varphi(j))$ be it's cost in the local optimum. Let $N^*(f^*) = \{j \mid \varphi^*(j) = f^*\}$ be the set of clients assigned to f^* in the optimal solution, and $N(f) = \{j \mid \varphi(j) = f\}$ be those assigned to f in the local optimum.

Lemma 3.22 *For each swap $(r, f^*) \in S$,*

$$\textsc{kMed}(F + f^* - r) - \textsc{kMed}(F) \leq \sum_{j \in N^*(f^*)} (O_j - A_j) + \sum_{j \in N(r)} 2\, O_j. \tag{3.6}$$

Proof: Consider the following candidate assignment of clients (which gives us an upper bound on the cost increase): map each client in $N^*(f^*)$ to f^*. For each client $j \in N(r) \setminus N^*(f^*)$, consider the figure below. Let the facility $\widehat{f^*} = \varphi^*(j)$: assign j to $\widehat{r} = \eta(\widehat{f^*})$, the closest facility in F to $\widehat{f}^*$. Note that by Fact 3.21 , $\widehat{r} \neq r$, and this is a valid new assignment. All other clients in $V \setminus (N(r) \cup N^*(f^*))$ stay assigned as they were in φ.


$\varphi(j)$
$\widehat{f} = \eta(\varphi^*(j))$
F
F^*
$\widehat{f}^* = \varphi^*(j)$
j

Note that for any client $j \in N^*(f^*)$, the change in cost is exactly $O_j - A_j$: summing over all these clients gives us the first term in the expression (3.6).

For any client $j \in N(r) \setminus N^*(f^*)$, the change in cost is

$$d(j, \widehat{r}) - d(j, r) \leq d(j, \widehat{f}^*) + d(\widehat{f}^*, \widehat{r}) - d(j, r) \tag{3.7}$$

$$\leq d(j, \widehat{f}^*) + d(\widehat{f}^*, r) - d(j, r) \tag{3.8}$$

$$\leq d(j, \widehat{f}^*) + d(j, \widehat{f}^*) = 2\,O_j. \tag{3.9}$$

with (3.7) and (3.9) following by the triangle inequality, and (3.8) using the fact that $\widehat{r}$ is the closest vertex in F to $\widehat{f}^*$. Summing up, the total change for all these clients is at most

$$\sum_{j \in N(r) \setminus N^*(f^*)} 2\,O_j \leq \sum_{j \in N(r)} 2\,O_j, \tag{3.10}$$

the inequality holding since we are adding in non-negative terms. This proves Lemma 3.22. ■

Note that summing (3.6) over all tuples in S, along with the fact that each $f^* \in F^*$ appears exactly once and each $r \in R \subseteq F$ appears at most twice gives us the simple proof of the following theorem.

Theorem 3.23 ([AGK+04]) *At a local minimum F, the cost $\textsc{kMed}(F) \leq 5 \cdot \textsc{kMed}(F^*)$.*

3.7 Conclusion

In this chapter, we studied the design and analysis of parallel approximation algorithms for facility-location problems, including facility location, k-center, k-median, and k-means. We presented several efficient algorithms, based on a diverse set of approximation algorithms techniques. The practicality of these algorithms is a matter pending experimental investigation.

Chapter 4

Parallel Max-Cut Using The Matrix Multiplicative Weights Method

In this chapter, we investigate a basic graph optimization problem known as the maximum cut (MaxCut) problem. This problem has been a catalyst in the development of many now-common techniques in both approximation algorithms and the theory of hardness of approximation. Given a graph $G = (V, E)$, the goal of the problem is to find a bipartition of the graph to maximize the number of edges crossing the partition. Mathematically,

$$\text{MaxCut}(G) = \max_{S \subseteq V(G)} |(S, \bar{S})|, \text{ where } (S, \bar{S}) \stackrel{\text{def}}{=} E(G) \cap (S \times (V(G) \setminus S)).$$

Sometimes, it is more convenient to work with a normalized objective. Let $\mathsf{val}_G(S, \bar{S}) \stackrel{\text{def}}{=} |(S, \bar{S})|/|E(G)|$ denote the fraction of the edges in the cut $(S, \bar{S})$. The MaxCut problem can equivalently be stated as finding a cut $(S, \bar{S})$ that maximizes $\mathsf{val}_G(S, \bar{S})$.

The MaxCut problem is known to be NP-complete [GJ79], but a number of surprisingly simple $\frac{1}{2}$-approximation algorithms are known. For example, a randomized algorithm that flips an unbiased coin to decide which part each node belongs to is a $\frac{1}{2}$-approximation. This algorithm runs in $O(|V|)$ time and can be derandomized to run in $O(|E|)$ time. Furthermore, the randomized algorithm is trivial to parallelize.

Goemans and Williamson (GW) [GW95] presented a major breakthrough that achieved an approximation ratio of $\alpha_{GW} \approx 0.878\cdots$ through a novel, yet simple, SDP rounding technique. Assuming the unique games conjecture, this approximation has been shown to be the best possible (see, e.g, [OW08] and the references therein for details). In the sequential setting, Klein and Lu [KL96] give an algorithm for general graphs with $\widetilde{O}(|V||E|)$ running time, and subsequently, Arora and Kale [AK07] give an algorithm for regular graphs with $\widetilde{O}(|E|)$ running time. It was not obvious how any of these SDP-based algorithms can be efficiently parallelized.

In this chapter, we give an SDP-based primal-dual algorithm that works for general graphs, building on the algorithm of Arora and Kale. Specifically, we prove the following theorem:

Theorem 4.1 *For a fixed constant $\varepsilon > 0$, there is a $(1-\varepsilon)\alpha_{GW}$-approximation algorithm for MaxCut on an unweighted graph with n nodes and m edges that runs in $O(m\log^7 n)$ work and $O(\log^5 n)$ depth.*

(We prove this theorem in Section 4.3)

MaxCut SDP

First, we begin by reviewing the MaxCut SDP. The MaxCut problem has an SDP relaxation based on a natural quadratic program formulation. It consists of n vectors (the rows of $\mathbf{X}$), each corresponding to a vertex of the input graph. Let $G = (V, E)$ be a graph and $\mathbf{L}$ be the Laplacian associated with it. The MaxCut SDP primal/dual pair (ignoring a scaling constant of $1/4$) is as follows.

$$\begin{array}{lll|lll} \text{Maximize} & \mathbf{L} \bullet \mathbf{X} & & \text{Minimize} & \mathbf{1}\cdot\mathbf{y} & \\ \text{Subj. to:} & X_{ii} \le 1 \text{ for } i = 1,\dots,|V| & & \text{Subj. to:} & \mathsf{diag}(\mathbf{y}) \succcurlyeq \mathbf{L}. & \\ & \mathbf{X} \succcurlyeq \mathbf{0} & & & \mathbf{y} \ge \mathbf{0}. & \end{array} \tag{4.1}$$

It is easy to see that $\mathsf{Tr}[\mathbf{X}] \le n$ in any primal feasible solution $\mathbf{X}$.

4.1 Arora-Kale Framework for Solving SDPs

In this section, we outline the primal-dual framework of Arora and Kale [AK07] for approximately solving semidefinite programs (SDPs) and describe how it can be implemented efficiently in parallel.

At STOC'07, Arora and Kale presented a primal-dual framework for approximately solving semidefinite programs (SDPs). Using a matrix multiplicative weights method, the framework allows for finding an approximate solution to an SDP through the help of an oracle (it should be compared with the framework of Plotkin, Shmoys, and Tardos [PST95] for packing/covering LPs and more recent work of Khandekar [Kha04]). More specifically, consider a general SDP pair, in which the primal has n^2 variables and m constraints:

$$\begin{array}{ll|ll} \text{Maximize} & \mathbf{C} \bullet \mathbf{X} & \text{Minimize} & \mathbf{b} \cdot \mathbf{y} \\ \text{Subj. to:} & \mathbf{A}_j \bullet \mathbf{X} \leq b_j \text{ for } j \in [m] & \text{Subj. to:} & \sum_{j=1}^m \mathbf{A}_j y_j \succcurlyeq \mathbf{C}. \\ & \mathbf{X} \succcurlyeq \mathbf{0} & & \mathbf{y} \geq \mathbf{0}. \end{array}$$

As with [AK07], we assume $\mathbf{A}_1 = \mathbf{I}$ and $b_1 = R$ for some $R \in \mathbb{R}_+$, which serves to bound the trace of the solution (i.e., $\mathsf{Tr}[\mathbf{X}] \leq R$). The framework solves the feasibility version of the problem; our algorithm will use binary search to derive the optimization version. Let α be the current guess for the SDP's optimum value. The framework starts with a trivial candidate (which might not be primal feasible) $\mathbf{X}^{(1)} = \frac{R}{n}\mathbf{I}$ and produces a series of candidate solutions $\mathbf{X}^{(2)}, \mathbf{X}^{(3)}, \ldots$ through the help of an oracle. For each $\mathbf{X}^{(t)}$, the oracle searches for a vector $\mathbf{y} \in \mathcal{D}_\alpha \stackrel{\text{def}}{=} \{\mathbf{y} \in \mathbb{R}_+^m : \mathbf{b} \cdot \mathbf{y} \leq \alpha\}$ such that

$$\sum_{j=1}^{m} [\mathbf{A}_j \bullet \mathbf{X}^{(t)}] y_j - [\mathbf{C} \bullet \mathbf{X}^{(t)}] \ \geq \ 0. \tag{4.2}$$

Arora and Kale show that if the oracle fails, scaling $\mathbf{X}^{(t)}$ suitably gives a primal feasible solution with value at least α [AK07]. If the oracle succeeds, $\mathbf{X}^{(t)}$ is either primal infeasible or has value at most α—and the vector $\mathbf{y}$ contains information useful for improving the candidate solution $\mathbf{X}^{(t)}$. Before we can describe the algorithm for updating $\mathbf{X}^{(t)}$, we need the following definition:

Definition 4.2 ((ℓ, ρ)-Boundedness Oracle) *For $0 \leq \ell \leq \rho$), an (ℓ, ρ)-bounded oracle is an oracle whose return value $\mathbf{y} \in \mathcal{D}_\alpha$ always satisfies either $\sum_j \mathbf{A}_j y_j - \mathbf{C} \in [-\ell, \rho]$ or $\sum_j \mathbf{A}_j y_j - \mathbf{C} \in [-\rho, \ell]$. Moreover, we say that the oracle has* width *ρ.*

The update procedure relies on the matrix multiplicative weights algorithm, which can be summarized as follows. For a fixed $\varepsilon_0 \leq \frac{1}{2}$ and $\mathbf{W}^{(1)} = \mathbf{I}$, we play a repeated "game" a number of times, where in iteration $t = 1, 2, \ldots$, the following steps are performed:

1. Produce a probability matrix $\mathbf{P}^{(t)} = \mathbf{W}^{(t)} / \mathsf{Tr}\left[\mathbf{W}^{(t)}\right]$,
2. Incur a loss matrix $\mathbf{M}^{(t)}$, and
3. Update the weight matrix as $\mathbf{W}^{(t+1)} = \exp(-\varepsilon_0 \sum_{t' \leq t} \mathbf{M}^{(t')})$.

Approximating Matrix Exponentials. For efficiency, we have to implement the matrix exponentials approximately. For the MaxCut algorithm, it suffices to approximate matrix exponentials using a Taylor's series expansion and dimensionality reduction, as described in [AK07]. But this means that the oracle also has to be approximate. We first describe a parallel implementation of the matrix exponentials approximation and proceed to discuss approximate oracles.

Arora and Kale apply the Johnson-Lindenstrauss (JL) projection to each $\mathbf{X}^{(t)} \in \mathbb{R}_+^{n\times n}$, resulting in n length-k vectors for $k = O(\log n/\varepsilon^2)$, where ε is a parameter that ensures pair-wise squared distance preservation up to $1 \pm \varepsilon$. This process is easily parallelized because it involves constructing an $n \times k$ projection matrix ($O(nk)$ work and $O(1)$ depth) and k sparse-matrix vector multiplies, all of which can be done in parallel, for a total of $O(mk)$ work and $O(\log(m+n))$ if $\mathbf{X}^{(t)} \in \mathbb{R}_+^{n\times n}$ has m nonzero entries.

Now each matrix $\mathbf{X}^{(t)} = \exp(-\varepsilon_0 \sum_{t'<t} \mathbf{M}^{(t)})$ needs not be sparse, but its approximation using Taylor's expansion has the nice properties that the matrices involved will be sufficiently sparse and for the precision required, the computation involves only a few matrix multiplies. First, it is easy to see that if $\mathbf{C}$ initially has p nonzeros, after t iterations $\sum_{t'\leq t} \mathbf{M}^{(t)}$ has at most $(p+n)t$ nonzeros. In our algorithm, as will soon be apparent, p will be the number of edges in the input graph plus the number of nodes, and t will always be $O(\text{polylog}(n))$. Second, note that to compute the JL projection, we only have to be able to find a vector $\mathbf{v}$ for each vector $\mathbf{u}$ (a row of the projection matrix) such that $\|\mathbf{X}^{(t)}\mathbf{u} - \mathbf{v}\|_2 \leq \tau \cdot \|\mathbf{X}^{(t)}\|$, where $\|\mathbf{X}^{(t)}\|$ denotes the spectral radius of $\mathbf{X}^{(t)}$ and τ is an accuracy parameter, which we will set to $n^{-O(1)}$. Finally, shown in the lemma below is the cost of computing one column (out of k) of the JL projection; this follows from Lemma 6 of [AK07].

Lemma 4.3 *Let an accuracy parameter τ be given. If $\mathbf{A} \in \mathbb{R}^{n\times n}$ has s nonzeros and spectral radius $\|\mathbf{A}\|$, then a vector $\mathbf{v}$ satisfying*

$$\|\exp(\mathbf{A})\mathbf{u} - \mathbf{v}\|_2 \leq \tau \cdot \|\exp(\mathbf{A})\|$$

can be found in $O(r(s+n))$ work and $O(r\log(s+n))$ depth, where $r = \max(\ln(\frac{1}{\tau}), e^2\|\mathbf{A}\|)$.

This holds because we are performing r sparse-matrix vector multiplies sequentially, where each is computable in $O(s+n)$ work and $O(\log(s+n))$ depth. As a consequence of this lemma, the total cost of JL projection (which results in n length-k vectors) is $O(rk(s+n))$ work and $O(r\log(s+n))$ depth, as each dimension of the output vectors can be computed independently. In our algorithm, τ will be $n^{-O(1)}$, $k = O(\log n)$, $s = O(m\log^2 n)$, and $\|\mathbf{A}\| = O(\log^2 n)$, so the work will be $O(m\log^5 n)$ and depth $O(\log^3 n)$, assuming $n \leq m \leq n^2$.

Approximate Oracles. While this approximation allows us to implement matrix exponentials efficiently, it does affect the guarantee we can expect from the oracle. As a result, the primal-dual framework has to be able to handle an oracle with weaker guarantees.

A δ-*approximate oracle* works with a less stringent guarantee than what is imposed by (4.2): it instead looks for a vector $\mathbf{y} \in \mathcal{D}_\alpha$ such that

$$\sum_{j=1}^{m} [\mathbf{A}_j \bullet \mathbf{X}^{(t)}] y_j - [\mathbf{C} \bullet \mathbf{X}^{(t)}] \;\geq\; -\alpha\delta. \tag{4.3}$$

Armed with these, we present the Arora-Kale framework in Algorithm 4.1.1. With $\tau = \frac{\delta\alpha}{48n^{5/2}R(\ell+\rho)}$ and a suitable setting of constants in the JL projection, the approximation turns an (ℓ,ρ)-bounded (exact) oracle into a $\frac{\delta}{3}$-approximate (ℓ,ρ)-bounded oracle, which suffices to achieve the following guarantees:

Theorem 4.4 (Arora-Kale [AK07]) *For a tolerance parameter $\delta > 0$, and a guess value α, the primal-dual algorithm using these approximations converges within $T = \frac{18\ell\rho R^2 \ln n}{\delta^2\alpha^2}$ rounds given an (ℓ,ρ)-bounded $\frac{\delta}{3}$-approximate oracle. Either it returns an $\mathbf{X}^{(t)}$ with value at least α or if it did not fail for T rounds, then there is a dual feasible solution with value at most $(1+\delta)\alpha$ given by the vector $\mathbf{y}$, where $y_1 = \frac{1}{T}\sum_{t=1}^{T} y_1^{(t)} + \delta\alpha/R$ and for $i > 1$, $y_i = \frac{1}{T}\sum_{t=1}^{T} y_i^{(t)}$.*

4.2 Graph Sparsification

We describe a graph sparsification procedure that will prove useful in our MaxCut algorithm. The underlying idea relies on standard sampling techniques that arose in various forms before.

Let $G = (V, E, w)$ be an n-node weighted graph. We obtain $\widehat{G}$ by sampling with replacement from G as follows. For $t = 1, 2, \ldots, T = \frac{2}{\lambda^2}(n + \ln n)$, pick an edge from G with probability proportional to its weight—and add this to $\widehat{G}$ with weight 1 (if the edge already exists, we simply increase its weight).

Extending the previous definition of $\mathsf{val}_G(S,\bar{S})$ to include weights, we let $\mathsf{val}_G(S,\bar{S}) := \sum_{ij\in(S,\bar{S})} w_{ij}/W(G)$, where $W(G) = \sum_{ij\in E(G)} w_{ij}$ is the total weight.

Algorithm 4.1.1 The Arora-Kale Primal-Dual Algorithm

Oracle: a $\frac{\delta}{3}$-approximate (ℓ, ρ)-bounded oracle

Let $T = \frac{18\ell\rho R^2 \ln n}{\delta^2\alpha^2}$ We will run the matrix multiplicative weights (MWU) algorithm with $\varepsilon_0 = \delta\alpha/3\ell R$ for $T =$ rounds. For $t = 1, \dots, T$,

1. Retrieve the probability matrix $\mathbf{P}^{(t)}$ from MWU and produce $\mathbf{X}^{(t)} = R\mathbf{P}^{(t)}$.
2. Run the $\frac{\delta}{3}$-approximate oracle with an approximation of $\mathbf{X}^{(t)}$, with $\tau = \frac{\delta\alpha}{48n^{5/2}R(\ell+\rho)}$. If the oracle fails, return $\mathbf{X}^{(t)}$; otherwise, we obtain $\mathbf{y}^{(t)}$.
3. Provide MWU with the following loss matrix:

$$\mathbf{M}^{(t)} = \frac{1}{\ell+\rho}\sum_{j=1}^{m}\mathbf{A}_j y_j^{(t)} - \mathbf{C} + \gamma^{(t)}\mathbf{I},$$

where

$$\gamma^{(t)} = \begin{cases} \ell & \text{if } \sum_j \mathbf{A}_j y_j - \mathbf{C} \in [-\ell, \rho] \\ -\ell & \text{if } \sum_j \mathbf{A}_j y_j - \mathbf{C} \in [-\rho, \ell] \end{cases}$$

Lemma 4.5 *For any cut $(S, \bar{S})$ in G, the sampled graph $\widehat{G}$ preserves the fractional cut value up to λ additive error* ***with high probability****. More precisely, with high probability,* $\mathsf{val}_{\widehat{G}}(S, \bar{S}) \in [\mathsf{val}_G(S, \bar{S}) - \lambda, \mathsf{val}_G(S, \bar{S}) + \lambda]$.

Proof: Consider the sequence of T edges sampled by this process and let e_t denote the t-th edge sampled. Note that the probability that an edge ij is sampled is $p_{ij} := w_{ij}/W$.

Now fix a cut $(S, \bar{S})$. Thus, for $t = 1, \dots, T$,

$$\mathbf{Pr}\left[e_t \in (S, \bar{S})\right] = \mathsf{val}_G(S, \bar{S}).$$

Furthermore, let $X_t = \mathbf{1}_{\{e_t \in (S, \bar{S})\}}$, so

$$\mathsf{val}_{\widehat{G}}(S, \bar{S}) = \frac{1}{T}\sum_t X_t.$$

It follows that $\mathbf{E}\left[\mathsf{val}_{\widehat{G}}(S, \bar{S})\right] = \mathsf{val}_G(S, \bar{S})$. By Chernoff bounds, we have that $\mathsf{val}_{\widehat{G}}(S, \bar{S})$ deviates additively by more than λ from $\mathsf{val}_G(S, \bar{S})$ with probability *at most* $2\exp(-2T\lambda^2)$. Since there are at most 2^n different cuts, setting $T = \frac{2}{\lambda^2}(n + \ln n)$ suffices for the lemma to hold with high probability. ■

4.3 Parallel MaxCut

We are now ready to describe the parallel MaxCut algorithm, which consists of two main components. First, we present a preprocessing routine which transforms an arbitrary graph with n nodes and m edges into a graph on at most $2m$ nodes *with the special property that none of the nodes have degree more than $O(\log n)$*. Then, we describe a parallel implementation of Arora and Kale's MaxCut algorithm for small-degree graphs.

4.3.1 Constructing Constant-Degree Graphs

The first ingredient we need is a parallel algorithm that turns an arbitrary graph into a graph of comparable size but with no large-degree node. Specificially, we prove the following lemma:

Lemma 4.6 *Fix $\lambda > 0$. For input an unweighted graph $G = (V, E)$ with n nodes and m edges, there is an algorithm `preprocess` that turns it into an $O(n + m)$-node, $O(\lambda^{-2}(n + m))$-edge (unweighted) graph $\widehat{H}$ with maximum degree at most $C_0 \log n$, where C_0 is a constant that depends on λ and the accuracy parameter λ satisfies*

$$\frac{\text{MaxCut}(\widehat{H})}{|E(\widehat{H})|} - \lambda \leq \frac{\text{MaxCut}(G)}{|E(G)|} \leq \frac{\text{MaxCut}(\widehat{H})}{|E(\widehat{H})|} + \lambda.$$

This process runs in $O(n + m)$ work and $O(\log(n + m))$ depth ***with high probability****.*

For this, we extract and parallelize a construction that was implicit in [Tre01] (which was developed for a different problem). Let $G = (V, E)$ be given. We will proceed by describing how to construct an auxiliary *weighted* graph H, which we never build in reality, and describe an efficient procedure to sample from it (without having to construct such a graph). To build H, for each node $u \in V(G)$, create $\deg(u)$ copies of u. Now for each edge $ij \in E(G)$, create $\deg(u)\deg(v)$ copies, each with weight $\frac{1}{\deg(u)\deg(v)}$ (i.e., we have one edge for each pair of u and v copies and the weights on them sum to 1). Thus, H will have $2|E(G)|$ nodes and $\sum_u(\deg(u))^2$ edges. Given the sheer size of H, constructing it explicitly would be prohibitively expensive. An important property of the graph H is that it maintains the MaxCut size in the following sense:

Lemma 4.7 *There is a cut $(S, \bar{S})$ in G with $\mathsf{val}_G(S, \bar{S}) \geq c$ if and only if there is a cut $(T, \bar{T})$ in H with $\mathsf{val}_H(T, \bar{T}) \geq c$.*

Proof: If $(S, \bar{S})$ is a cut in G with $\mathsf{val}_G(S, \bar{S}) \geq c$, we construct T by including for each $u \in S$ all of u's copies. By construction, $(T, \bar{T})$ cuts the same fraction of edges in H. Conversely, we give a probabilistic argument. Let $(T, \bar{T})$ be a cut in H such that $\mathsf{val}_H(T, \bar{T}) = \kappa \geq c$. For each $u \in V(G)$, let $m(u)$ be the number of copies of u that are present in T and $p(u) = m(u)/\deg_G(u)$. Construct S as follows: include u in S with probability $p(u)$. It is easy to verify that $\mathbf{E}\left[\mathsf{val}_G(S, \bar{S})\right] = \kappa$, proving the existence of such a cut in G. ■

This proof also sheds light on how we can recover a cut in G from a cut we find in H. However, at this point, it may seem that we have not made any progress at all because H could have high-degree vertices and we would not be able to apply the algorithm of Arora and Kale. Here is where the graph sparsification idea from Section 4.2 comes in handy. Lemma 4.5 says that if we sample $\xi = \frac{2}{\lambda^2}(N + \ln N)$ times from an N-node graph H, we have a new graph $\widehat{H}$ with at most $\xi = O(N)$ edges and hence constant average degree. Note that H has at most $O(n + m)$ nodes.

Next, we will show how to sample efficiently from H while ensuring that $\widehat{H}$ has its maximum degree bounded by $O(\log n)$. The idea is to observe that we can sample an edge from H by first picking an edge uv uniformly at random from G and then deciding which $\deg(u)$ and $\deg(v)$ copies of u and v to put this to. In this way, an edge $xy \in E(H)$ with weight $\frac{1}{\deg(x)\deg(y)}$ has probability exactly $\frac{1}{\deg(x)\deg(y)}$ of being chosen, yielding the same probability distribution as sampling explicitly from H. Moreover, in this way, we can construct $\widehat{H}$ in parallel in $O(\xi)$ work and $O(\log N)$ depth.

To make sure $\widehat{H}$ does not have any high-degree node, we generate and check each candidate graph until the graph does not contain a node with degree more than $O(\log n)$. As the next claim shows, with high probability, we have to try at most a constant number of times.

Claim 4.8 *There is a constant C_0 depending on λ such that the sampling procedure constructs a graph with maximum degree at most $C_0 \log n$ with probability exceeding $1 - n^{-2}$.*

Proof: Consider a node x in $\widehat{H}$. For $t = 1, \ldots, \xi$, let Y_t indicate whether our t-th sampled edge is incident to x, so $\deg_{\widehat{H}}(x) \leq \sum_t Y_t$. But $\mathbf{E}\left[Y_t\right] = \frac{\deg_G(u)}{|E(G)|}\frac{1}{\deg_G(u)}$, where u is the "mother" node of x. Thus, $\mathbf{E}\left[\sum_t Y_t\right] = \frac{\xi}{|E(G)|} = O(1/\lambda^2)$. Since these edges are sampled independently, the claim then follows from Chernoff bounds (and union bounds taken over the nodes), like the balls-and-bins analysis. ■

Combining these claims completes the proof of Lemma 4.6.

4.3.2 MaxCut on Constant-Degree Graphs

We revisit the MaxCut algorithm of Arora and Kale [AK07] with an eye towards parallelization. (Basic familiarity with their algorithm is assumed.) Consider the SDP in (4.1). We will be working with the graph $\widehat{H}$, so for the remainder of this section, let $N = |V(\widehat{H})|$ and $M = |E(\widehat{H})|$. Their MaxCut algorithm relies on the observation that on an unweighted graph $\widehat{H}$ with maximum degree $\Delta = O(\log n)$, there is an $(O(\Delta), O(\Delta))$-bounded oracle and that the SDP optimal value lies between $2M$ and $6M$. Therefore, performing binary search on this range to accuracy ε requires at most $O(\log \frac{1}{\varepsilon})$ rounds. Furthermore, as observed earlier, the trace bound of this SDP is $R = N$. Hence, by Theorem 4.4, the primal-dual framework converges to accuracy δ within $O(\frac{1}{\delta^2} \log^2 N)$ rounds. Within each round, the dominant cost is from computing the approximate Cholesky decomposition using a Taylor's expansion and JL projection; this[1] takes $O(M \log^5 N)$ work and $O(\log^3 N)$ depth per round.

Their oracle algorithm is easy to parallelize. It involves basic operations that can be performed in $O(\log(N + M))$ depth and $O(N + M)$ work and computing $\mathbf{L} \bullet \mathbf{X}$, which can be done in $O(\log(N + M))$ depth and $O(M \log N)$ work, as follows: Note first that we are given as input to the oracle length-$O(\log N)$ vectors $\mathbf{v}_1, \ldots, \mathbf{v}_N$ that approximate the Cholesky decomposition of $\mathbf{X}$ (as described in Section 4.1). Then, we observe that $\mathbf{L}_{\widehat{H}} \bullet \mathbf{X} = \sum_{ij \in E(\widehat{H})} \|\mathbf{v}_i - \mathbf{v}_j\|^2$; for each edge $ij \in E(\widehat{H})$, we can compute $\|\mathbf{v}_i - \mathbf{v}_j\|^2$ in $O(\log \log N)$ depth and $O(\log N)$ work, for a total of $O(\log \log N)$ depth and $O(M \log N)$ work (the depth term does not increase because all these computations can be done in parallel). Finally, we sum up these values for all the edges, in $O(\log M)$ depth and $O(M)$ work. Thus, computing $\mathbf{L}_{\widehat{H}} \bullet \mathbf{X}$ can be done in $O(\log(N + M))$ depth and $O(M \log N)$ work, as desired.

Following the argument in Arora and Kale [AK07], in conjuction with the parallel implementation outlined above, we have:

Theorem 4.9 (Adapted from [AK07]) *Let $\widehat{H}$ be an unweighted graph on N nodes and M edges with $O(\log n)$ maximum degree. The MaxCut SDP (4.1) can be approximated to arbitrary constant, but fixed, accuracy in $O(M \log^7 N)$ work and $O(\log^5 N)$ depth.*

Rounding the SDP

Given N length-$O(\log N)$ vectors, $\mathbf{v}_1, \ldots, \mathbf{v}_N$, with SDP value $\alpha \geq (1 - \delta)\alpha^*$, where α^* is the optimum, the rounding procedure proceeds as usual: pick a random unit vector $\mathbf{n}$ (from

[1]Since the number of iterations is bounded by $O(\log^2 N)$, the technical condition that the spectral radius of each $\mathbf{A}$ has to be small is met. Indeed, $\|\mathbf{A}\| \leq O(\log^2 N)$.

a spherical symmetric distribution), compute the dot-products $\mathbf{n} \cdot \mathbf{v}_i$ for all i, and derive a cut by putting the positive ones on one side and the negative ones on the other. This rounding step takes $O(\log N)$ depth and $O(N \log N)$ work. Furthermore, Goemans and Williamson show that the cut obtained in this way has size, in expectation, at least $\alpha_{GW} \cdot \alpha$, which is at least $(1-\delta)\alpha_{GW}$ times the size of the optimal cut.

Finally, we need to translate this cut back to a cut in the graph we started off. Using Lemmas 4.6 and 4.7, we have the following corollary:

Corollary 4.10 *If $(T, \bar{T})$ is a cut in $\widehat{H}$ such that $\mathsf{val}_{\widehat{H}}(T, \bar{T}) = \kappa$, then generating a cut $(S, \bar{S})$ using the randomized algorithm in Lemma 4.7 satisfies* $\mathbf{E}\left[\mathsf{val}_G(S, \bar{S})\right] \geq \kappa - \lambda - o(1)$.

This step can be performed within the same cost as the rounding step. For large enough graph and appropriate settings of λ, δ, and ε, we have that the cut $(S, \bar{S})$ in G satisfies

$$\mathbf{E}\left[\mathsf{val}_G(S, \bar{S})\right] \geq (1-\delta)\alpha_{GW} \cdot \mathsf{opt} - \lambda - o(1) \geq (1-\varepsilon)\alpha_{GW},$$

where opt denotes the fraction of the edges cut in an optimal solution. Combining the results in this section and observing that $N \leq |V(G)| + 2|E(G)|$ and $M = O(N)$, we have completed the proof of the main theorem (Theorem 4.1) of this chapter.

Chapter 5

Maximal Nearly Independent Set and Applications

Set cover is one of the most fundamental and well-studied problems in optimization and approximation algorithms. This problem and its variants have a wide variety of applications in the real world, including locating warehouses, testing faults, scheduling crews on airlines, and allocating wavelength in wireless communication. Let $\mathcal{U}$ be a set of n ground elements, $\mathcal{F}$ be a collection of subsets of $\mathcal{U}$ covering $\mathcal{U}$ (i.e., $\cup_{S\in\mathcal{F}} S = \mathcal{U}$), and $c\colon \mathcal{F} \to \mathbb{R}_+$ a cost function. The *set cover problem* is to find the cheapest collection of sets $\mathcal{A} \subseteq \mathcal{F}$ such that $\cup_{S\in\mathcal{A}} S = \mathcal{U}$, where the cost of the solution $\mathcal{A}$ is specified by $c(\mathcal{A}) = \sum_{S\in\mathcal{A}} c(S)$. Unweighted set cover (all weights are equal) appeared as one of the 21 problems Karp identified as NP-complete in 1972 [Kar72]. Two years later, Johnson [Joh74] proved that the simple greedy method gives an approximation that is at most a factor $H_n = \sum_{k=1}^{n} \frac{1}{k}$ from optimal. Subsequently, Chvátal [Chv79a] proved the same approximation bounds for the weighted case. These results are complemented by a matching hardness result: Feige [Fei98] showed that unless $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{O(\log\log n)})$, set cover cannot be approximated in polynomial time with a ratio better than $(1 - o(1)) \ln n$. This essentially shows that the greedy algorithm is optimal.

Not only is greedy set cover optimal but it also gives an extremely simple $O(M)$ time algorithm for the unweighted case and $O(M \log M)$ time for the weighted case, where $M \geq n$ is the sum of the sizes of the sets. In addition, ideas similar to greedy set cover have been successfully applied to max k-cover, min-sum set cover, k-center, and facility location, generally leading to optimal or good-quality, yet simple, approximation algorithms.

From a parallelization point of view, however, the greedy method is in general difficult to parallelize, because at each step, only the highest-utility option is chosen and every subsequent step is likely to depend on the preceding one. Berger, Rompel, and Shor [BRS94] (BRS) showed that the greedy set cover algorithm can be "approximately" parallelized by bucketing[1] utility values (in this case, the number of new elements covered per unit cost) by factors of $(1+\varepsilon)$ and processing sets within a bucket in parallel. Furthermore, the number of buckets can be kept to $O(\log n)$ by preprocessing. However, deciding which sets within each bucket to choose requires some care: although at a given time, many sets might have utility values within a factor of $(1+\varepsilon)$ of the current best option, the sets taken together might not cover as many *unique* elements as their utility values imply—shared elements can be counted towards only one of the sets. BRS developed a technique to subselect within a bucket by first further bucketing by cost, then set size, and finally element degree, and then randomly selecting sets with an appropriate probability. This leads to an $O(\log^5 M)$-depth and $O(m \log^4 M)$-work randomized algorithm, giving a $((1+\varepsilon)H_n)$-approximation on a PRAM. Rajagopalan and Vazirani [RV98] improved the depth to $O(\log^3(Mn))$ with work $O(M \log^2 M)$ but at the cost of a factor of two in the approximation (essentially a factor-$2(1+\varepsilon)H_n$ approximation).

In comparison to their sequential counterparts, none of these previous set-cover algorithms are work efficient—their work is asymptotically more than the time for the optimal sequential algorithm.[2] Work efficiency is important since it allows an algorithm to be applied efficiently to both a modest number of processors (one being the most modest) and a larger number. Even with a larger number of processors, work-efficient algorithms limit the amount of resources used and hence presumably the cost of the computation.

Summary of Results

In this chapter, we abstract out the most important component of the bucketing approach, which we refer to as Maximal Nearly Independent Set (MaNIS), and develop an $O(m)$ work and $O(\log^2 m)$ depth algorithm for an input graph with m edges, on the EREW PRAM. The MaNIS problem is to find a subset of sets such that they are nearly independent (their elements do not overlap too much), and maximal (no set can be added without introducing too

[1]The bucketing approach has also been used for other algorithms such as vertex cover [KVY94] and metric facility location [BT10].

[2]We note that the sequential time for a weighted $(1+\varepsilon)H_n$-approximation for set-cover is $O(M)$ when using bucketing.

much overlap). Since we have to look at the input, which has size $O(m)$, the algorithm is work efficient. The MaNIS abstraction allows us to reasonably easily apply it to several set-cover-like problems. In particular, we develop the following work-efficient approximation algorithms:

— *Set cover.* We develop an $O(M)$ work, $O(\log^3 M)$ depth (parallel time) algorithm with approximation ratio $(1+\varepsilon)H_n$. For the unweighted case, the same algorithm gives a $(1+\varepsilon)(1+\ln(n/\mathsf{opt}))$ approximation where opt is the optimal set-cover cost.

— *Max cover.* We develop an $O(M)$ work, $O(\log^3 M)$ depth prefix-optimal algorithm with approximation ratio $(1-1/e-\varepsilon)$. This significantly improves the work bounds over a recent result [CKT10].

— *Min-sum set cover.* We develop an $O(M)$ work, $O(\log^3 M)$ depth algorithm with an approximation ratio of $(4+\varepsilon)H_n$. We know of no other RNC parallel approximation algorithms for this problem.

— *Asymmetric k-center.* We develop an $O(p(k+\log^* n))$ work, $O(k\log n+\log^3 n\log^* n)$ depth algorithm with approximation ratio $O(\log^* n)$, where $p = n(n-1)/2$ is the size of the table of distances between elements. The algorithm is based on the sequential algorithm of Panigrahy and Vishwanathan [PV98] and we know of no other RNC parallel approximation algorithms for this problem.

— *Metric facility location.* We develop an $O(p\log p)$ work, $O(\log^4 p)$ depth algorithm with approximation ratio $(1.861+\varepsilon)$, where $p = |F|\times|C|$ is the size of the distance table. The algorithm is based on the greedy algorithm of Jain et al. [JMM+03] and improves on the approximation ratio of $(3+\varepsilon)$ for the best previous RNC algorithm [BT10].

All these algorithms run on a CRCW PRAM but rely on only a few primitives discussed in the next section, and thus are easily portable to other models.

5.1 Preliminaries and Notation

The results in this chapter use both the EREW (Exclusive Read Exclusive Write) and CRCW (Concurrent Read Concurrent Write) variants of the PRAM, and for the CRCW, we assume an arbitrary value is written. For an input of size n, we assume that every memory word has $O(\log n)$ bits. In our analysis, we are primarily concerned with minimizing the work while achieving polylogarithmic depth and less concerned with polylogarithmic factors in the depth since such measures are typically not robust across models. All algorithms we develop are in NC or RNC, so they have polylogarithmic depth.

The algorithms in this chapter are all based on a bipartite graph $G = (A \cup B, E), E \subseteq A \times B$. In set cover, for example, we use A to represent the subsets $\mathcal{F}$ and B for the ground elements $\mathcal{U}$. In addition to operating over the vertices and edges of the graph, the algorithms need to copy a value from each vertex (on either side) to its incident edges, need to "sum" values from the incident edges of each vertex using a binary associative operator, and given $A' \subseteq A$ and $B' \subseteq B$ need to *subselect* the graph $G' = (A' \cup B', (A' \times B') \cap E)$.

For analyzing bounds, we assume that G is represented in a form of adjacency array we refer to as the *packed representation* of G. In this representation, the vertices in A and B and the edges in E are each stored contiguously, and each vertex has a pointer to a contiguous array of pointers to its incident edges. With this representation, all the operations mentioned in the previous paragraph can be implemented using standard techniques in $O(|G|)$ work and $O(\log |G|)$ depth on an EREW PRAM, where $|G| = |A| + |B| + |E|$. The set-cover algorithm also needs the following operation for constructing the packed representation.

Lemma 5.1 *Given a bipartite graph $G = (A \cup B, E)$ represented as an array of $a \in A$, each with a pointer to an array of integer identifiers for its neighbors in B, the packed representation of G can be generated with $O(|G|)$ work and $O(\log^2 |G|)$ depth (both w.h.p.) on a* CRCW PRAM.

Proof: We note that the statement of the lemma allows for the integer identifiers to be sparse and possibly much larger than $|B|$. To implement the operation use duplicate elimination over the identifiers for B to get a unique representative for each $b \in B$ and give these representatives contiguous integer labels in the range $[|B|]$. This can be done with hashing in randomized $O(|G|)$-work $O(\log^2 |G|)$-depth [BM98]. Now that the labels for B are bounded by $[|B|]$ we can use a bounded integer sort [RR89] to collect all edges pointing to the same $b \in B$ and generate the adjacency arrays for the vertices in B in randomized $O(n)$ work and $O(\log n)$ depth on a (arbitrary) CRCW PRAM. ■

We will also use the following.

Lemma 5.2 *If $y_1, \dots, y_n \in (0, 1]$ are drawn independently such that* **Pr** $\left[x_i \in \left(\frac{j-1}{n}, \frac{j}{n}\right]\right] \leq \frac{1}{n}$ *for all $i, j = 1, \dots, n$, then the keys $y_1, \dots, y_n$ can be sorted in expected $O(n)$ work and $O(\log n)$ depth on an* CRCW PRAM

Proof: Use parallel radix sort to bucket the keys into $B_1, \ldots, B_n$, where B_j contains keys between $\left(\frac{j-1}{n}, \frac{j}{n}\right]$. This requires $O(n)$ work and $O(\log n)$ depth. Then, for each B_i, in parallel, we can sort the elements in the bucket in $O(|B_i|^2)$ work $O(|B_i|)$ depth using, for example, a parallel implementation of the insertion sort algorithm. The work to sort these buckets is $\mathbf{E}\left[\sum_i |B_i|^2\right] \leq 2n$. Furthermore, balls-and-bins analysis shows that for all i, $|B_i| \leq O(\log n)$ with high probability. Thus, the depth of the sorting part is bounded by $\mathbf{E}\left[\max_i |B_i|\right] \leq O(\log n)$. ■

5.2 Maximal Nearly Independent Set (MaNIS)

We motivate the study of Maximal Nearly Independent Set (MaNIS) by revisiting existing parallel algorithms for set cover. The basic idea is as follows. These algorithms define a notion of utility—the number of new elements covered per unit cost—for each available option (set). Each iteration then involves identifying and working on the remaining sets that have utility within a $(1+\varepsilon)$ factor of the current best utility value—and for fast progress, requires that the best option after an iteration has utility at most a $(1+\varepsilon)$ factor smaller than before. Among the sets meeting the criterion, deciding which ones to include in the final solution is non-trivial. Selecting any one of these sets leads to an approximation ratio within $(1+\varepsilon)$ of the strictly greedy algorithm but may not meet the fast progress requirement. Including all of them altogether leads to arbitrarily bad bounds on the approximation ratio (many sets are likely to share ground elements) but does ensure fast progress. To meet both requirements, we would like to select a "maximal" collection of sets that have small, bounded overlap—if a set is left unchosen, its utility must have dropped sufficiently. This leads to the following graph problem formulation, where the input bipartite graph models the interference between sets.

Definition 5.3 ($(\boldsymbol{\varepsilon}, \boldsymbol{\delta})$-MaNIS) *Let $\varepsilon, \delta > 0$. Given a bipartite graph $G = (A \cup B, E)$, we say that a set $J \subseteq A$ is a (ε, δ)* maximal nearly independent set*, or (ε, δ)-MaNIS, if*

(1) Nearly Independent. *The chosen options do not interfere much with each other, i.e.,*

$$|N(J)| \quad \geq \quad (1-\delta-\varepsilon)\sum_{a \in J} |N(a)|.$$

(2) Maximal. *The unchosen options have significant overlaps with the chosen options, i.e., for all $a \in A \setminus J$,*

$$|N(a) \setminus N(J)| < (1-\varepsilon)|N(a)|$$

The first condition in this MaNIS definition only provides guarantees on average—it ensures that *on average* each chosen option does not overlap much with each other. It is often desirable to have a stronger guarantee that provides assurance on a per-option basis. This motivates the following strengthened definition, which implies the previous definition.

Definition 5.4 (Ranked (ε, δ)-MaNIS) *Let $\varepsilon, \delta > 0$. Given a bipartite graph $G = (A \cup B, E)$, we say that a set $J = \{s_1, s_2, \dots, s_k\} \subseteq A$ is a* ranked (ε, δ) maximal nearly independent set*, or a ranked (ε, δ)-MaNIS for short, if*

(1) Nearly Independent. *There is an ordering (not part of the MaNIS solution) $s_1, s_2, \dots, s_k$ such that each chosen option s_i is almost completely independent of $s_1, s_2, \dots, s_{i-1}$, i.e., for all $i = 1, \dots, k$,*

$$|N(s_i) \setminus N(\{s_1, s_2, \dots, s_{i-1}\})| \geq (1 - \delta - \varepsilon)|N(s_i)|.$$

(2) Maximal. *The unchosen options have significant overlaps with the chosen options, i.e., for all $a \in A \setminus J$,*

$$|N(a) \setminus N(J)| < (1 - \varepsilon)|N(a)|.$$

Under this definition, an algorithm for ranked MaNIS only has to return a set J but not the ordering. Furthermore, the following fact is easy to verify:

Fact 5.5 *If J is a ranked (ε, δ)-MaNIS, then every $J' \subseteq J$ satisfies $|N(J')| \geq (1 - \delta - \varepsilon) \sum_{j \in J'} |N(j)|$.*

Connection with previous work: Both versions of MaNIS can be seen as a generalization of maximal independent set (MIS). Indeed, when $\delta = \varepsilon = 0$, the problem is the maximal set packing problem, which can be solved using a maximal independent set algorithm [KW85, Lub86], albeit with $O(\log n)$ more work than the simple sequential algorithm that solves both versions of MaNIS in $O(|E|)$ sequential time.

Embedded in existing parallel set-cover algorithms are steps that can be extracted to compute MaNIS. We obtain from Berger et al. [BRS94] (henceforth, the BRS algorithm) an RNC^4 algorithm for computing $(\varepsilon, 8\varepsilon)$-MaNIS in $O(|E| \log^3 n)$ work. Similarly, we extract from Rajagopalan and Vazirani [RV98] (henceforth, the RV algorithm) an RNC^2 algorithm for computing *ranked* $(\varepsilon^2, 1 - \frac{1}{2(1+\varepsilon)} - \varepsilon^2)$-MaNIS in $O(|E| \log |E|)$ work.

Unfortunately, neither of the existing algorithms, as analyzed, is work efficient. In addition, the existing analysis of the RV algorithm places a restriction on δ: even when ε is arbitrarily close to 0, we cannot have δ below $\frac{1}{2}$.

Algorithm 5.2.1 $\texttt{MaNIS}_{(\varepsilon,3\varepsilon)}$ — a parallel algorithm for computing ranked $(\varepsilon, 3\varepsilon)$-MaNIS

Input: a bipartite graph $G = (A \cup B, E)$.

Output: $J \subseteq A$ satisfying Definition 5.4.

```
Initialize $G^{(0)} = (A^{(0)} \cup B^{(0)}, E^{(0)}) = (A \cup B, E)$, and
for each $a \in A$, $D(a) = |N_{G^{(0)}}(a)|$.
Set $t = 0$. Repeat the following steps until $A^{(t)}$ is empty:

  1. For $a \in A^{(t)}$, randomly pick $x_a \in_R [0, 1]$
  2. For $b \in B^{(t)}$, let $\varphi^{(t)}(b)$ be $b$'s neighbor with maximum $x_a$
  3. Pick vertices of $A^{(t)}$ chosen by sufficiently many in $B^{(t)}$:
       $J^{(t)} = \Big\{a \in A^{(t)} \,\Big|\, \sum_{b \in B^{(t)}} \mathbf{1}_{\{\varphi^{(t)}(b)=a\}} \geq (1 - 4\varepsilon)D(a)\Big\}$.
  4. Update the graph by removing $J$ and its neighbors, and elements of $A^{(t)}$ with too few remaining neighbors:
       $B^{(t+1)} = B^{(t)} \setminus N_{G^{(t)}}(J^{(t)})$
       $A^{(t+1)} = \{a \in A^{(t)} \setminus J^{(t)} : |N_{G^{(t)}}(a) \cap B^{(t+1)}| \geq (1 - \varepsilon)D(a)\}$
       $E^{(t+1)} = E^{(t)} \cap (A^{(t+1)} \times B^{(t+1)})$
  5. $t = t + 1$

Finally, return $J = J^{(0)} \cup \cdots \cup J^{(t-1)}$.
```

5.2.1 Linear-Work Ranked MaNIS

We present an algorithm for the ranked MaNIS problem. Our algorithm is inspired by the RV algorithm. Not only is the algorithm work efficient but also it removes the $\frac{1}{2}$ restriction on δ, matching and surpassing the guarantees given by previous algorithms. To obtain these bounds, we need a new analysis that differs from that of the RV algorithm. Our algorithm can be modified to compute ranked (ε, δ)-MaNIS for any $0 < \varepsilon < \delta$ in essentially the same work and depth bounds (with worse constants); however, for the sake of presentation, we settle for the following theorem:

Theorem 5.6 (Ranked MaNIS) *Fix $\varepsilon > 0$. For a bipartite graph $G = (A \cup B, E)$ in packed representation there exists a randomized* EREW PRAM algorithm $\texttt{MaNIS}_{(\varepsilon,3\varepsilon)}(G)$ that produces a ranked $(\varepsilon, 3\varepsilon)$-MaNIS in $O(|E|)$ expected work and $O(\log^2 |E|)$ expected depth.

Presented in Algorithm 5.2.1 is an algorithm for computing ranked MaNIS. To understand this algorithm, we will first consider a natural sequential algorithm for $(\varepsilon, 3\varepsilon)$-MaNIS—and discuss modifications that have led us to the parallel version. To compute MaNIS, we could first pick an ordering of the vertices of A and consider them in this order: for each $a \in A$, if a has at least $(1-4\varepsilon)D(a)$ neighbors, we add a to the output and delete its neighbors; otherwise, set it aside. Thus, every vertex added certainly satisfies the nearly-independent requirement. Furthermore, if a vertex is not added, its degree must have dropped below $(1-\varepsilon)D(a)$, which ensures the maximality condition.

Algorithm 5.2.1 achieves parallelism in two ways. First, we adapt the selection process so that multiple vertices can be chosen together at the same time. Unlike the sequential algorithm, the parallel version can decide whether to include a vertex $a \in A$ without knowing the outcomes of the preceding vertices. This is done by making the inclusion condition more conservative: Assign each $b \in B$ to the first $a \in A$ in the chosen ordering—regardless of whether a will be included in the solution. Then, for each $a \in A$, include it in the solution if enough of its neighbors are assigned to it. This step is highly parallel and ensures that every vertex added satisfies the nearly-independent requirement. Unfortunately, this process by itself may miss vertices that must be included.

Second, we repeat the selection process until no more vertices can be selected but ensure that the number of iterations is small. As the analysis below shows, a random permutation allows the algorithm to remove a constant fraction of the edges, making sure that it will finish in a logarithmic number of iterations. Note that unlike before, the multiple iterations make it necessary to distinguish between the original degree of a vertex, $D(a)$, and its degree in the current iteration (which we denote by $\deg(a)$ in the proof). Furthermore, we need an clean-up step after each iteration to eliminate vertices that are already maximal so that they will not hamper progress in subsequent rounds.

Running Time Analysis: To prove the work and depth bounds, consider the potential function

$$\Phi^{(t)} \stackrel{\text{def}}{=} \sum_{a \in A^{(t)}} |N_{G^{(t)}}(a)|,$$

which counts the number of remaining edges. The following lemma shows that sufficient progress is made in each step:

Lemma 5.7 *For $t \geq 0$, $\mathbf{E}\left[\Phi^{(t+1)}\right] \leq (1-c)\,\Phi^{(t)}$, where $c = \frac{1}{4}\varepsilon^2(1-\varepsilon)$.*

Before proceeding with the proof, we offer a high-level sketch. We say a vertex $a \in A^{(t)}$ deletes an edge (a', b) if $a \in J^{(t)}$ and $\varphi^{(t)}(b) = a$. In essence, the proof shows that for $a \in A^{(t)}$, the expected number of edges a deletes, denoted by Δ_a in the proof, is proportional to the degree of a. If a has few neighbors, it suffices to consider the probability that all neighbors select a. Otherwise, the proof separates the neighbors of a into high- and low-

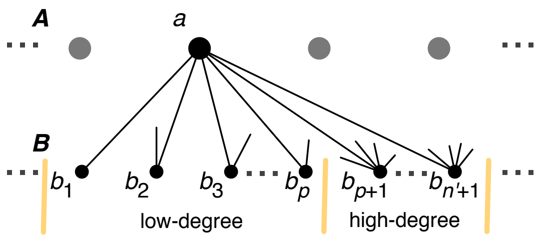


Figure 5.1: MaNIS analysis: for each $a \in A^{(t)}$, order its neighbors so that $N_{G^{(t)}}(a) = \{b_1, \ldots, b_{n'}\}$ and $\deg(b_1) \leq \deg(b_2) \leq \cdots \leq \deg(b_{n'})$, where $n' = \deg_{G^{(t)}}(a)$.

degree groups and analyzes Δ_a by averaging over possible values of x_a. In particular, it considers a y_a (i.e., $1 - \varepsilon / \deg(b_p)$ in the proof) such that for all $x_a \geq y_a$, there are likely sufficiently many low-degree neighbors that select a to ensure with constant probability that a is in $J^{(t)}$. Then, the proof shows that there is sufficient contribution to Δ_a from just the high-degree neighbors and just when $x_a \geq y_a$ (that is when a is likely in $J^{(t)}$). This is formalized in the proof below.

Proof: Consider an iteration t. Let $\deg(x) = \deg_{G^{(t)}}(x)$ and $\Delta_a = \mathbf{1}_{\{a \in J^{(t)}\}} \sum_{b:\varphi^{(t)}(b)=a} \deg(b)$. Thus, when a is included in $J^{(t)}$, Δ_a is the sum of the degrees of all neighbors of a that are assigned to a (by $\varphi^{(t)}$). It is zero otherwise if $a \notin J^{(t)}$. Since $\varphi^{(t)} : B^{(t)} \to A^{(t)}$ maps each $b \in B^{(t)}$ to a unique element in $A^{(t)}$, the sum of Δ_a over a is a lower bound on the number edges we lose in this round. That is,

$$\Phi^{(t)} - \Phi^{(t+1)} \geq \sum_{a \in A^{(t)}} \Delta_a,$$

so it suffices to show that for all $a \in A^{(t)}$, $\mathbf{E}\left[\Delta_a\right] \geq c \cdot \deg(a)$.

Let $a \in A^{(t)}$ be given and assume WLOG that $N_{G^{(t)}}(a) = \{b_1, \ldots, b_{n'}\}$ such that $\deg(b_1) \leq \deg(b_2) \leq \cdots \leq \deg(b_{n'})$. (as shown in Figure 5.1). Now consider the following cases:

— *Case 1. a has only a few neighbors:* Suppose $n' < \frac{2}{\varepsilon}$. Let $\mathcal{E}_1$ be the event that $x_a = \max\{x_{a'} : a' \in N_{G^{(t)}}(N_{G^{(t)}}(A^{(t)}))\}$. Then, $\mathcal{E}_1$ implies that (1) $a \in J^{(t)}$ and (2) $\varphi^{(t)}(b_{n'}) = a$. Therefore,

$$\mathbf{E}\left[\Delta_a\right] \geq \mathbf{Pr}\left[\mathcal{E}_1\right] \cdot \deg(b_{n'}) \geq \tfrac{1}{n'} \geq c \cdot \deg(a),$$

because $|N_{G^{(t)}}(N_{G^{(t)}}(A^{(t)}))| \leq n' \cdot \deg(b_{n'})$ and $n' = \deg(a) < 2/\varepsilon$.

— *Case 2. a has many neighbors*, i.e., $n' \geq \frac{2}{\varepsilon}$. Partition the neighbors of a into low- and high- degree elements as follows. Let $p = \lfloor (1 - \varepsilon) \deg(a) \rfloor$, $L(a) = \{b_1, \ldots, b_p\}$, and

$H(a) = \{b_{p+1}, \ldots, b_{n'}\}$. To complete the proof for this case, we rely on the following claim.

Claim 5.8 *Let* $\mathsf{select}_a^{(t)} = \{b \in B^{(t)} : \varphi^{(t)}(b) = a\}$*, and* $\mathcal{E}_2$ *be the event that* $|L(a) \setminus \mathsf{select}_a^{(t)}| \leq 2\varepsilon|L(a)|$*. Then, (i) for* $\gamma \leq \varepsilon/\deg(b_p)$*,* $\mathbf{Pr}\left[\mathcal{E}_2|x_a = 1-\gamma\right] \geq \frac{1}{2}$*; and (ii) for* $b \in H(a)$ *and* $\gamma \leq \varepsilon/\deg(b)$*,* $\mathbf{Pr}\left[\varphi^{(t)}(b) = a|\mathcal{E}_2, x_a = 1-\gamma\right] \geq 1-\varepsilon$*.*

Note that $\mathcal{E}_2$ implies $|\mathsf{select}_a^{(t)}| \geq n' - \varepsilon n' - 2\varepsilon n' \geq (1-4\varepsilon)D(a)$, because $n' \geq (1-\varepsilon)D(a)$. This in turn means that $\mathcal{E}_2$ implies $a \in J^{(t)}$. Applying the claim, we establish

$$\begin{aligned}
\mathbf{E}\left[\Delta_a\right] &\geq \sum_{b \in H(a)} \deg(b)\, \mathbf{Pr}\left[\mathcal{E}_2 \wedge \varphi^{(t)}(b) = a\right] \\
&\geq \sum_{b \in H(a)} \int_{\gamma=0}^{\frac{\varepsilon}{\deg(b)}} \deg(b)\, \mathbf{Pr}\left[\mathcal{E}_2 | x_a = 1-\gamma\right] \\
&\qquad \mathbf{Pr}\left[\varphi^{(t)}(b) = a \middle| \mathcal{E}_2, x_a = 1-\gamma\right] d\gamma \\
&\geq \sum_{b \in H(a)} \varepsilon \frac{1}{2}(1-\varepsilon) \\
&\geq c \cdot \deg(a),
\end{aligned}$$

where the final step follows because $|H(a)| \geq \varepsilon n' \geq 1$. ■

Proof of Claim 5.8: To prove (i), let $X \overset{\text{def}}{=} |L(a) \setminus \mathsf{select}_a^{(t)}| = \sum_{j \in L(a)} \mathbf{1}_{\{j \notin \mathsf{select}_a^{(t)}\}}$, so

$$\mathbf{E}\left[X|x_a = 1-\gamma\right] = \sum_{j \in L(a)} \mathbf{Pr}\left[j \notin \mathsf{select}_a^{(t)} | x_a = 1-\gamma\right].$$

Then, note that $j \in \mathsf{select}_a^{(t)}$ iff. $x_a = \max\{x_i : i \in N_{G^{(t)}}(j)\}$. Thus, for $j \in L(a)$,

$$\mathbf{Pr}\left[j \notin \mathsf{select}_a^{(t)} | x_a = 1-\gamma\right] \leq 1 - \left(1 - \tfrac{\varepsilon}{\deg(b_p)}\right)^{\deg(j)} \leq \varepsilon,$$

and so $\mathbf{E}\left[X|x_a = 1-\gamma\right] \leq \varepsilon|L(a)|$. By Markov's inequality, we have $\mathbf{Pr}\left[\mathcal{E}_2|x_a = 1-\gamma\right] \geq 1 - \frac{\varepsilon}{2\varepsilon} = \frac{1}{2}$.

We will now prove (ii). Consider that for $i \in N_{G^{(t)}}(b) \setminus \{a\}$, $\mathbf{Pr}\left[x_i > x_a|\mathcal{E}_2, x_a = 1-\gamma\right] \leq \gamma$. Union bounds give

$$\begin{aligned}
\mathbf{Pr}\left[\varphi^{(t)}(b) = a|\mathcal{E}_2, x_a = 1-\gamma\right] &\geq 1 - \sum_{i \in N_{G^{(t)}}(b) \setminus \{a\}} \gamma \\
&\geq 1 - \varepsilon.
\end{aligned}$$

■

Each iteration can be implemented in $O(\Phi^{(t)})$ work and $O(\log \Phi^{(t)})$ depth on an EREW PRAM since beyond trivial parallel application and some summations, the iteration only involves the operations on the packed representation of G discussed in Section 5.1. Since $\Phi^{(t)}$ decreases geometrically (in expectation), the bounds follow. Algorithm 5.2.1 as described selects random reals x_a between 0 and 1. But it is sufficient for each a to use a random integer with $O(\log n)$ bits such that w.h.p., there are no collisions. In fact, since the x_a's are only compared, it suffices to use a random permutation over A since the distribution over the ranking would be the same.

5.3 Linear-Work Set Cover

As our first example, in this section, we apply MaNIS to parallelize a greedy approximation algorithm for weighted set cover. Specifically, we prove the following theorem:

Theorem 5.9 *Fix $0 < \varepsilon < \frac{1}{2}$. For a set system $(\mathcal{U}, \mathcal{F})$, where $|\mathcal{U}| = n$, there is a randomized $(1+\varepsilon)H_n$-approximation for (weighted) set cover that runs in $O(M)$ expected work and $O(\log^3 M)$ expected depth on a* CRCW PRAM, where $M = \sum_{S \in \mathcal{F}} |S|$.

We describe a parallel greedy approximation algorithm for set cover in Algorithm 5.3.1. To motivate the algorithm, we discuss three ideas crucial for transforming the standard greedy set-cover algorithm into a linear-work algorithm with substantial parallelism: (1) approximate greedy through bucketing, (2) prebucketing and lazy bucket transfer, and (3) subselection via MaNIS. Despite the presence of some of these ideas in previous work, it is the combination of our improved MaNIS algorithm and careful lazy bucket transfer that is responsible for better work and approximation bounds.

Like in previous algorithms, bucketing creates opportunities for parallelism at the round level, by grouping together sets by their coverage-per-unit-cost values in powers of $(1-\varepsilon)$. Consequently, there will be at most $O(\log_{1+\varepsilon} \rho)$ buckets (also rounds), where ρ is the ratio between the largest and the smallest coverage-per-unit-cost values. This, however, raises several questions (which we resolve by ideas (2) and (3)).

First, *how can we make ρ small and keep the contents of the relevant buckets "fresh" in linear work?* As detailed in Lemma 5.10, the algorithm relies on a subroutine `Prebucket` that first preprocesses the input to keep ρ polynomially bounded by setting aside certain "cheap" sets that will be included in the final solution by default and throwing away sets that will never

Algorithm 5.3.1 `SetCover` — parallel greedy set cover.

Input: a set cover instance $(\mathcal{U}, \mathcal{F}, c)$.

Output: a collection of sets covering the ground elements.

```
i.   Let $\gamma = \max_{e \in \mathcal{U}} \min_{S \in \mathcal{F}} c(S)$,
     $M = \sum_{S \in \mathcal{F}} |S|$,
     $T = \log_{1/(1-\varepsilon)}(M^3/\varepsilon)$,
     and $\beta = \frac{M^2}{\varepsilon \cdot \gamma}$.
ii.  Let $(\mathcal{A}; A_0, \ldots, A_T) = \texttt{Prebucket}(\mathcal{U}, \mathcal{F}, c)$ and $\mathcal{U}_0 = \mathcal{U}$
iii. For $t = 0, \ldots, T$, perform the following steps:
     1. Remove deleted elements from sets in this bucket:
        $A'_t = \{S \cap \mathcal{U}_t : S \in A_t\}$
     2. Only keep sets that still belong in this bucket:
        $A''_t = \{S \in A'_t : |S|/c(S) > \beta \cdot (1-\varepsilon)^{t+1}\}$.
     3. Select a maximal nearly independent set from the bucket:
        $J_t = \texttt{MaNIS}_{(\varepsilon, 3\varepsilon)}(A''_t)$.
     4. Remove elements covered by $J_t$:
        $\mathcal{U}_{t+1} = \mathcal{U}_t \setminus X_t$ where $X_t = \cup_{S \in J_t} S$
     5. Move remaining sets to the next bucket:
        $A_{t+1} = A_{t+1} \cup (A'_t \setminus J_t)$
iv.  Finally, return $\mathcal{A} \cup J_0 \cup \cdots \cup J_T$.
```

be used in the solution. It then classifies the sets into buckets $A_0, A_1, \ldots A_T$ by their utility; however, this initial bucketing will be stale as the algorithm progresses. While we cannot afford to reclassify the sets in every round, it suffices to maintain an invariant that each set in $S \in A_i$ satisfies $|S \cap \mathcal{U}_t|/c(S) \leq \beta \cdot (1-\varepsilon)^i$. Furthermore, we make sure that the bucket that contains the current best option is fresh—and move the sets that do not belong there accordingly.

Second, *what to do with the sets in each bucket to satisfy both the bucket invariant and the desired approximation ratio?* This is where we apply MaNIS. As previously discussed in Section 5.2, MaNIS allows the algorithm to choose nearly non-overlapping sets, which helps bound the approximation guarantees and ensures that sets which MaNIS leaves out can be moved to the next bucket and satisfy the bucket invariant.

In the following lemma and proof, we use the definitions for γ, M, T, and β from Algorithm 5.3.1. Furthermore, let opt denote the optimal cost.

Lemma 5.10 *There is an algorithm* `Prebucket` *that takes as input a set system* $(\mathcal{U}, \mathcal{F}, c)$ *and produces a set* $\mathcal{A}$ *such that* $c(\mathcal{A}) \leq \varepsilon \cdot \text{opt}$ *and buckets* $A_0, \ldots, A_T$ *such that*

1. *for each $S \in \mathcal{F} \setminus \mathcal{A}$, either S costs more than $M\gamma$ or $S \in A_i$ for which $|S|/c(S) \in (\beta \cdot (1-\varepsilon)^{i+1}, \beta \cdot (1-\varepsilon)^i]$.*
2. *there exists a set cover solution costing at most* opt *using sets from $\mathcal{A} \cup A_0 \cup A_1 \cup \cdots \cup A_T$;*
3. *the algorithm runs in $O(M)$ work and $O(\log M)$ depth on a* CRCW PRAM.

Proof: We rely on the following bounds on opt [RV98]: $\gamma \leq \text{opt} \leq M\gamma$. Two things are clear as a consequence: (i) if $c(S) \leq \varepsilon \cdot \frac{\gamma}{M}$, S can be included in $\mathcal{A}$, yielding a total cost at most $\varepsilon\gamma \leq \varepsilon \cdot \text{opt}$. (ii) if $c(S) > M\gamma$, then S can be discarded (S is not part of any optimal solution).

Thus, we are left with sets costing between $\varepsilon \cdot \frac{\gamma}{M}$ and $M\gamma$. Compute $|S|/c(S)$ for each remaining set S, in parallel, and store S in A_i such that $|S|/c(S) \in (\beta \cdot (1-\varepsilon)^{i+1}, \beta \cdot (1-\varepsilon)^i]$. We know that $1/(M\gamma) \leq |S|/c(S) \leq M^2/(\varepsilon\gamma) = \beta$, so the buckets are numbered between $i = 0$ and $i = \log_{1/(1-\varepsilon)}(M^3/\varepsilon) = T$.

Computing M, γ, and $|S|/c(S)$ for all sets S can be done in $O(M)$ work and $O(\log M)$ depth using parallel sums. Once each set knows which bucket it belongs to, a stable integer sort over integers in the range $[O(\log M)]$ can be used to collect them into buckets with the same work and depth bounds [RR89]. ■

Approximation Guarantees: We follow a standard proof in Vazirani [Vaz01]. It should be noted that although we do not mention LPs here, the proof given below is similar in spirit to the dual-fitting proof presented by Rajagopalan and Vazirani [RV98]. Let $p_t = \frac{1}{\beta}(1-\varepsilon)^{-(t+1)}$. For each $e \in \mathcal{U}$, if e is covered in iteration t, define the price of this element to be $p(e) = p_t$. That is, every element covered in this iteration has the same price p_t. Now, Step 2 ensures that if $S \in A''_t$ can cover e, then $c(S)/|S| \leq p(e)$, where $|S|$ is the size of S after Step 2 in iteration t. Let $X_t = \cup_{S \in J_t} S$ be the set of elements covered in iteration t. The near independent property of $(\varepsilon, 3\varepsilon)$-MaNIS indicates that $|X_t| = |N(J_t)| \geq (1-4\varepsilon)\sum_{S \in J_t} N(S)$, where $N(\cdot)$ here is the neighborhood set in A''_t. Thus, $c(J_0 \cup \cdots \cup J_T)$ can be written as

$$\sum_t \sum_{S \in J_t} \frac{c(S)}{|S|} \cdot |S| \leq \sum_t p_t \sum_{S \in J_t} |S| \leq \frac{1}{1-4\varepsilon} \sum_{e \in \mathcal{U}} p(e).$$

Let $\mathcal{O}^*$ be any optimal solution. Consider a set $S^* \in \mathcal{O}^*$. Since all buckets $t' < t$ are empty, we know that $p_t \leq \frac{1}{1-\varepsilon} \min_{S \in A''_t} \frac{c(S)}{|S|}$. Furthermore, for each $e \in S^*$, let t_e denote the iteration in which e was covered. By greedy properties (as argued in [RV98, Vaz01]),

$$\sum_{e \in S^*} \min_{S \in G''_{t_e}} \frac{c(S)}{|S|} \leq \left(1 + \frac{1}{2} + \frac{1}{3} + \dots \frac{1}{|S^*|}\right) c(S^*).$$

Hence, $c(J_0 \cup \dots \cup J_T) \leq \frac{1}{1-5\varepsilon} \sum_{S^* \in \mathcal{O}^*} H_{|S^*|} c(S^*) \leq \frac{1}{1-5\varepsilon} H_n \cdot \mathsf{opt} \leq (1+\varepsilon') H_n \cdot \mathsf{opt}$ (using $\varepsilon = O(\varepsilon')$), as promised.

Implementation and Work and Depth Bounds: To analyze the cost of the algorithm, we need to be more specific about the representation of all structures that are used. We assume the sets $S \in \mathcal{F}$ are given unique integer identifiers $[|\mathcal{F}|]$, and similarly for the elements $e \in \mathcal{U}$. Each set keeps a pointer to an array of identifiers for its elements, and each bucket keeps a pointer to an array of identifiers for its sets. The sets can shrink over time as elements are filtered out in Step 1 of each iteration of the algorithm. We keep a Boolean array indicating which of the elements from $\mathcal{U}$ remain in $\mathcal{U}_t$. Let $\mathcal{A}^{(t)}$ be the snapshot of A_t at the beginning of iteration t of Algorithm 5.3.1, and $M_t = \sum_{S \in \mathcal{A}^{(t)}} |S|$.

Claim: Iteration t of Algorithm 5.3.1 can be accomplished in expected $O(M_t)$ work and $O(\log^2 M_t)$ depth on the randomized CRCW PRAM.

Steps 1 and 2 use simple filtering on arrays of total length $O(M_t)$, which can be done with prefix sums. Step 3 requires converting adjacency arrays for each set in A''_t to the packed representation needed by MaNIS. The indices of the elements might be sparse, but this conversion can be done using Lemma 5.1. The cost of this conversion, as well as the cost of running MaNIS, is within the claimed bounds. Step 4 and 5 just involve setting flags, a filter, and an append, all on arrays of length $O(M_t)$.

Now there are $O(\log M)$ iterations, and `Prebucket` has depth $O(\log M)$, so the overall depth is bounded by $O(\log^3 M)$. To prove the work bound, we will derive a bound on $\sum_t M_t$. We note that every time a set is moved from one bucket to the next its size decreases by a constant factor, and hence the total work attributed to each set is proportional to its original size. More formally, we have the following claim:

Claim: If $S \in \mathcal{A}^{(t)}$, $|S| \leq c(S) \cdot \beta \cdot (1-\varepsilon)^t$.

This claim can be shown inductively: For any set $S \in \mathcal{F}$, `Prebucket` guarantees that the bucket that S went into satisfies the claim. Following that, this set can be shrunk and moved (Steps 1, 2, and 5). It is easy to check that the claim is satisfied (by noting Steps 2 and 5's criteria and that sets not chosen by MaNIS are shrunk by an ε fraction).

By this claim, the total sum $\sum_t M_t$ is at most

$$\sum_t \sum_{S\in\mathcal{A}^{(t)}} c(S)\cdot\beta\cdot(1-\varepsilon)^t \le \sum_{S\in\mathcal{F}} \frac{1}{\varepsilon}\cdot c(S)\beta\cdot(1-\varepsilon)^{t_S},$$

where t_S is the bucket index of S in the initial bucketing. Furthermore, Lemma 5.10 indicates that $|S| \ge c(S)\cdot\beta\cdot(1-\varepsilon)^{t_S+1}$, showing that $\sum_t M_t = O(\frac{1}{\varepsilon}M)$ as $\varepsilon \le \frac{1}{2}$. Since the work on each step is proportional to M_t, the overall work is $O(\frac{1}{\varepsilon}M)$.

5.4 Set Covering Variants

Building on the `SetCover` algorithm from the previous section, we describe simple changes to the algorithm or the analysis that result in solutions to variants of set cover. In this section, *we will be working with unweighted set cover.*

Ordered vs. Unordered. We would like to develop algorithms for prefix-optimal max cover and min-sum set cover, using our set-cover algorithm; however, unlike set cover, these problems require an ordering on the chosen sets—not just an unordered collection. As we now describe, minimal changes to the `SetCover` algorithm will enable it to output an ordered sequence of sets which closely approximate the greedy behavior. Specially, we will give an algorithm with the following property[3]: Let $\mathcal{T} \subseteq \mathcal{U}$ be given. Suppose there exist ℓ sets covering $\mathcal{T}$, and our parallel algorithm outputs an ordered collection $S_1, \dots, S_p$ covering $\mathcal{U}$, then

Lemma 5.11 *For any $i \le p$, the number of elements in $\mathcal{T}$ freshly covered by S_i, i.e., $|S_i \cap R_i|$, is at least $(1-5\varepsilon)|R_i|/\ell$, where $R_i = \mathcal{T} \setminus (\cup_{j<i} S_j)$ contains the elements of $\mathcal{T}$ that remain uncovered after choosing $S_1, \dots, S_{i-1}$.*

We modify the `SetCover` algorithm as follows. Make `MaNIS` returns a totally ordered sequence, by sorting each $J^{(t)}$ by their x_a's values and stringing together the sorted sequences $J^{(0)}, J^{(1)}, \dots$; this can be done in the same work-depth bounds (Lemma 5.2) in CRCW. Further, modify `SetCover` so that (1) `Prebucket` only buckets the sets (but will not throw away sets nor eagerly include some of them) and (2) its Step iv. returns a concatenated sequence, instead of a union. Again, this does not change the work-depth bound but outputs an ordered sequence.

[3]This is the analog of the following fact from the sequential greedy algorithm [You95, PV98]: if there exist ℓ sets covering $\mathcal{T}$, and greedy picks sets $\chi_1, \dots, \chi_p$ (in that order) covering $\mathcal{U}$, then for $i \le p$, the number of elements in $\mathcal{T}$ freshly covered by χ_i is at least $|R_i|/\ell$, where $R_i = \mathcal{T} \setminus (\cup_{j<i}\chi_j)$.

Next, we show that the output sequence has the claimed property by proving the following technical claim (variables here refer to those in Algorithm 5.3.1). Lemma 5.11 is a direct sequence of this claim (note that the sets we output come from $J_0, J_1, \ldots$ in that order).

Claim 5.12 *For all $t \geq 0$, if $\widehat{J}_t \subseteq J_t$ and $\widehat{X}_t = \cup_{S \in \widehat{J}_t} S$, then $|\widehat{X}_t \cap \mathcal{T}| \geq (1-5\varepsilon) \cdot |\widehat{J}_t| \cdot |Q_t|/\ell$, where $Q_t = \mathcal{T} \setminus (\cup_{t'<t} X_{t'})$.*

Proof: Let $t \geq 0$. By our assumption, there exist ℓ sets that fully cover Q_t. An averaging argument shows that there must be a single set, among the remaining sets, with a coverage ratio of at least $|Q_t|/\ell$. Since at the beginning of iteration t, we have $A_{t'} = \emptyset$ for $t' < t$, it follows that $\tau_t \geq |Q_t|/\ell$, where $\tau_t = \beta \cdot (1-\varepsilon)^t$. Furthermore, all sets $S \in A''_t$ have the property that $|S| \geq \tau_t(1-\varepsilon)$. Furthermore, Fact 5.5 guarantees that $\widehat{J}_t$ covers, among $\mathcal{T}$, at least $|N(\widehat{J}_t)| \geq (1-4\varepsilon)\sum_{j \in \widehat{J}_t} |N(j)| \geq (1-4\varepsilon)(1-\varepsilon)\tau_t|\widehat{J}_t| \geq (1-5\varepsilon)|\widehat{J}_t||Q_t|/\ell$, proving the lemma. ■

5.4.1 Max Cover

The max k-cover problem takes as input an integer $k > 0$ and a set system (generally unweighted), and the goal is to find k sets that cover as many elements as possible. The sequential greedy algorithm gives a $(1-1/e)$-approximation, which is tight assuming standard complexity assumptions. In the parallel setting, previous parallel set-covering algorithms do not directly give $(1-\frac{1}{e}-\varepsilon)$-approximation. But in the related MapReduce model, Chierichetti et al. [CKT10] give a $1-1/e-\varepsilon$-approximation, which gives rise to a $O(m \log^3 M)$-work algorithm in PRAM, where $M = \sum_{S \in \mathcal{F}} |S|$. This is not work efficient compared to the greedy algorithm, which runs in at most $O(M \log M)$ time for any k.

In this section, we give a factor-$(1-\frac{1}{e}-\varepsilon)$ prefix optimal algorithm for max cover. As in Chierichetti et al. [CKT10], we say that a sequence of sets $S_1, S_2, \ldots, S_p$ covering the whole ground elements is *factor-σ prefix optimal* if for all $k \leq p$, $|\cup_{i \leq k} S_i| \geq \sigma \cdot \mathrm{opt}_k$, where opt_k denotes the optimal coverage using k sets. More specifically, we prove the following theorem:

Theorem 5.13 *Fix $0 < \varepsilon < \frac{1}{2}$. There is a factor-$(1-\frac{1}{e}-\varepsilon)$ prefix optimal algorithm the max cover problem requiring $O(M)$ work and $O(\log^3 M)$ depth, where $M = \sum_{S \in \mathcal{F}} |S|$.*

Proof: Use the algorithm from Lemma 5.11, so the work-depth bounds follow immediately from set cover. To argue prefix optimality, let k be given and $OPT_k \subseteq \mathcal{F}$ be an optimal max k-cover solution. Applying Lemma 5.11 with $\mathcal{T} = OPT_k$ gives that $|R_{i+1}| \leq |R_i|(1 - \frac{1}{k}(1-5\varepsilon))$ and $|R_1| = |OPT_k|$. Also, we know that using $S_1, \ldots, S_k$, we will have covered at least $OPT_k - |R_{k+1}|$ elements of OPT_k. By unfolding the recurrence, we have $OPT_k - |R_{k+1}| \geq OPT_k - OPT_k \cdot \exp\{-(1-5\varepsilon)\}$. Setting $\varepsilon = \frac{e}{5}\varepsilon'$ completes the proof. ■

5.4.2 Special Case: Unweighted Set Cover

When the sets all have the same cost, we can derive a slightly different and stronger form of approximation guarantees for the same algorithm. We apply this bound to derive guarantees for asymmetric k-center in Section 5.4.4. The following corollary can be derived from Lemma 5.11 in a manner similar to the max-cover proof; we omit the proof in the interest of space.

Corollary 5.14 *Let $0 < \varepsilon \leq \frac{1}{2}$. For an unweighted set cover instance, set cover can be approximated with cost at most $\mathsf{opt}(1+\varepsilon)(1+\ln(n/\mathsf{opt}))$, where opt is the cost of the optimal set cover solution.*

5.4.3 Min-Sum Set Cover

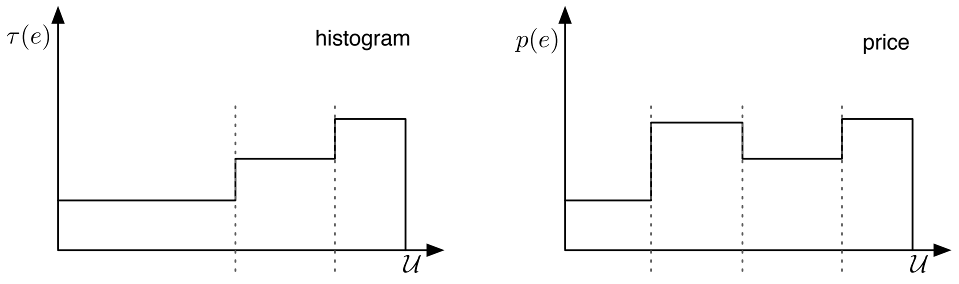
Another important and well-studied set covering problem is the min-sum set cover problem: given a set system $(\mathcal{U}, \mathcal{F})$, the goal is to find a sequence $S_1, \ldots, S_{n'}$ to minimize the cost $cost(\langle S_1, \ldots, S_{n'}\rangle) \stackrel{\text{def}}{=} \sum_{e \in \mathcal{U}} \tau(e)$, where $\tau(e) \stackrel{\text{def}}{=} \min\{i : e \in S_i\}$. Feige et al. [FLT04] (also implicit in Bar-Noy et al. [BNBH$^+$98]) showed that the standard set cover algorithm gives a 4-approximation, which is optimal unless $\mathsf{P} = \mathsf{NP}$. The following theorem shows that this carries over to our parallel set cover algorithm:

Theorem 5.15 *Fix $0 < \varepsilon \leq \frac{1}{2}$. There is a parallel $(4+\varepsilon)$-approximation algorithm for the min-sum set cover problem that runs in $O(M)$ work and $O(\log^3 M)$ depth.*

Proof: Consider the modified algorithm in Lemma 5.11. Suppose it outputs a sequence of sets $\mathsf{Alg} = \langle S_1, S_2, \ldots, S_{n'}\rangle$ covering $\mathcal{U}$, and an optimal solution is $\mathcal{O}^* = \langle O_1, \ldots, O_q\rangle$.

Let $\alpha_i = S_i \setminus (\cup_{j<i} S_j)$ denote the elements freshly covered by S_i and $\beta_i = \mathcal{U} \setminus (\cup_{j<i} S_j)$ be the elements not covered by $S_1, \ldots, S_{i-1}$. Thus, $|\beta_i| = |\mathcal{U}| - \sum_{j<i} |\alpha_j|$. Following

Feige et al. [FLT04], the cost of our solution is $cost(\mathsf{Alg}) = \sum_{i>0} i \cdot |\alpha_i| = \sum_{i>0} |\beta_i|$, which can be rewritten as $\sum_{i>0} \sum_{e \in \alpha_i} \frac{|\beta_i|}{|\alpha_i|} = \sum_{e \in \mathcal{U}} p(e)$, where the price $p(e) = \frac{|\beta_i|}{|\alpha_i|}$ for i such that $e \in \alpha_i$. We will depict and argue about these costs pictorially as follows. First, the "histogram" diagram (below) is made up of $|\mathcal{U}|$ horizontal columns, ordered from left to right in the order the optimal solution covers them. The height of column $e \in \mathcal{U}$ is its $\tau(e)$ in the optimal solution. Additionally, the "price" diagram is also made up of $|\mathcal{U}|$ columns, though ordered from left to right in the order our solution covers them; the height of column e is $p(e)$.



We can easily check that (1) the histogram curve has area $\mathsf{opt} = cost(\mathcal{O}^*)$ and (2) the price curve has area $cost(\mathsf{Alg})$. We will show that shrinking the price diagram by 2 horizontally and θ vertically (θ to be chosen later) allows it to lie completely inside the histogram when they are aligned on the bottom-right corner. Let $p = (x, y)$ be a point inside (or on) the price diagram. Suppose p lies in the column $e \in \alpha_i$, so $y \le p(e) = |\beta_i|/|\alpha_i|$—and p is at most $|\beta_i|$ from the right.

When shrunk, p will have height $h = p(e)/\theta$ and width—the distance from the right end—$r = \frac{1}{2}|\beta_i|$. We estimate how many elements inside β_i are covered by the optimal solution using its first h sets. Of all the sets in $\mathcal{F}$, there exists a set S such that $|S \cap \beta_i| \ge |O_j^* \cap \beta_i|$ for all $j < i$.

Arguing similarly to previous proofs in this section, we have that $|\alpha_i| \ge (1 - 5\varepsilon)|S|$, so at this time, the optimal algorithm could have covered at most $h \cdot \frac{1}{1-5\varepsilon}|\alpha_i|$. Setting $\theta = \frac{2}{1-5\varepsilon}$ gives that the first h sets of $\mathcal{O}^*$ will leave $|\beta_i| - \frac{1}{2}|\beta_i| \ge \frac{1}{2}|\beta_i| = r$ elements of β_i remaining. Therefore, the scaled version of p lies inside the histogram, proving that the algorithm is a 2θ-approximation. By setting $\varepsilon = \frac{1}{40}\varepsilon'$, we have $2\theta = \frac{4}{1-5\varepsilon} \le 4 + \varepsilon'$, which completes the proof. ■

5.4.4 Application: Asymmetric k-Center

Building on the set cover algorithm we just developed, we present an algorithm for asymmetric k-center. The input is an integer $k > 0$ and a distance function $d\colon V \times V \to \mathbb{R}_+$, where V is a set of n vertices; the goal is to find a set $F \subseteq V$ of k centers that minimizes the objective $\max_{j\in V} \min_{i\in F} d(i,j)$, where $d(x,y)$, which needs not be symmetric, denotes the distance from x to y. The distance d, however, is assumed to satisfy the triangle inequality, i.e., $d(x,y) \le d(x,z) + d(z,y)$. For symmetric d, there is a 2-approximation for both the sequential [HS85, Gon85] and parallel settings [BT10]. This result is optimal assuming $\mathsf{P} \neq \mathsf{NP}$. However, when d is not symmetric—hence the name asymmetric k-center, there is a $O(\log^* n)$-approximation in the sequential setting, which is also optimal unless $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{O(\log\log n)})$ [CGH+05], but nothing was previously known for the parallel setting.

In this section, we develop a parallel factor-$O(\log^* n)$ algorithm for this problem, based on the (sequential) algorithm of Panigrahy and Vishwanathan [PV98]. Thier algorithm consists of two phases: recursive cover and find-and-halve.

Recursive Cover for Asymmetric k-Center. The recursive cover algorithm of Panigrahy and Vishwanathan [PV98] (shown below) is easy to parallelize given the set cover routine from Corollary 5.14. Here, V is the input set of vertices.

```
Set $A_0 = A$ and $i = 0$. While $|A_i| > 2k$, repeat the following:
  1. Construct a set cover instance $(\mathcal{U}, \mathcal{F})$, where $\mathcal{U} = A_i$ and $\mathcal{F} = \{S_1, \ldots, S_{|V|}\}$ such that
     $S_x = \{y \in A_i : d(x,y) \le r\}$.
  2. $B = \texttt{SetCover}(\mathcal{U}, \mathcal{F})$.
  3. $A_{i+1} = B \cap A$ and $i = i + 1$.
```

Let $n = |A|$. Assuming d is given as a distance matrix, Step 1 takes $O(n^2)$ work and $O(\log n)$ depth (to generate the packed representation). In $O(n^2)$ work and $O(\log^3 n)$ depth, Steps 2 and 3 can be implemented using the set-cover algorithm (Section 5.4.2) and standard techniques. Following [PV98]'s analysis, we know the number of iterations is at most $O(\log^* n)$. Therefore, this recursive cover requires $O(n^2 \log^* n)$ work and $O(\log^* n \log^3 n)$ depth.

Next, the find-and-halve phase will be run sequentially beyond trivial parallization; in the section that follows, we give some evidence why parallelizing it might be difficult. Combining these components, we have the following theorem:

Theorem 5.16 *Let $\varepsilon > 0$. There is a $O(n^2 \cdot (k + \log^* n))$-work $O(k \cdot \log n + \log^3 n \log^3 n)$-depth factor-$O(\log^* n)$ approximation algorithm for the asymmetric k-center problem*

Note that the algorithm performs essentially the same work as the sequential one. Furthermore, for $k \le \log^{O(1)} n$, this is an RNC algorithm. As suggested in [PV98], the recursive cover procedure alone yields a bicriteria approximation, in which the solution consists of $2k$ centers and costs at most $O(\log^* n)$ more than the optimal cost. This bicriteria approximation has $O(\log^{O(1)} n)$ depth for any k.

5.4.5 Find-and-Halve is P-Complete

The find-and-halve procedure is a crucial preprocessing for recursive cover. This procedure is responsible for pruning out center capturing vertices (CCV). To describe CCV and find-and-halve, we need some definitions: let $d(u \to v)$ be the distance from u to v, and R be a fixed constant. Define $N^+(v) := \{u : d(v \to u) \le R\}$ and $N^-(v) := \{u : d(u \to v) \le R\}$. A vertex v is a *center capturing vertex* (CCV) if $N^-(v) \subseteq N^+(v)$. The sequential algorithm is simple: for a given distance parameter R, while there exists a CCV vertex v, remove v as well as all vertices w such that $d(v \to w) \le 2R$.

We show that the find-and-halve procedure is P-complete by showing an NC reduction from NOR-CVP, which is known to be P-complete [JáJ92, GHR95]. In particular, we prove the following theorem:

Theorem 5.17 *There is an* NC reduction from NOR-CVP to an instance of asymmetric k-center such that if I is any sequence produced by the find-and-halve procedure, the gate g_i outputs `true` if and only if $v_i^{(0)}$ appears in I. Therefore, the problem of computing such a sequence is P-complete.

Before describing the reduction, let us review the NOR-CVP problem:

Definition 5.18 (NOR-CVP) *Give a Boolean circuit consisting of gates $g_1, \ldots, g_n$ such that g_i is either an input equal to `true` or $g_i =$ NOR(g_k, g_ℓ) for $k < \ell < i$, the* NOR-CVP *problem is to determine whether C outputs `true`.*

The Reduction. For each input gate g_i, we create a node $v_i^{(0)}$. For each non-input gate $g_i =$ NOR(g_k, g_ℓ), we build the following 4-node gadget (Figure 5.2, left): $v_i^{(0)}, v_i^{(1)}, v_i^{(2)}$, and $v_i^{(3)}$ with arcs $v_i^{(0)} \to v_i^{(2)}, v_i^{(1)} \to v_i^{(0)}, v_i^{(2)} \to v_i^{(1)}, v_i^{(2)} \to v_i^{(3)}, v_i^{(3)} \to v_i^{(1)}$; all these arcs have length R. Furthermore, there are also arcs connecting between gadgets. For $g_i = \mathsf{NOR}(g_k, g_\ell)$, we have arcs $v_k^{(0)} \to v_i^{(1)}$ with length $R - \varepsilon$ and $v_k^{(0)} \to v_i^{(0)}$ with length $R + \varepsilon$, and symmetrically, we have $v_\ell^{(0)} \to v_i^{(3)}$ with length $R - \varepsilon$ and $v_\ell^{(0)} \to v_i^{(0)}$ with length $R + \varepsilon$. This reduction results in an asymmetric k-center instance, in fact a weighted directed graph. We give an example of such reductions in Figure 5.2. The distance $d(u \to v)$ is the shortest-path distance from $u \to v$ in this graph; therefore, this (directional) distance function satisfies the triangle inequality.

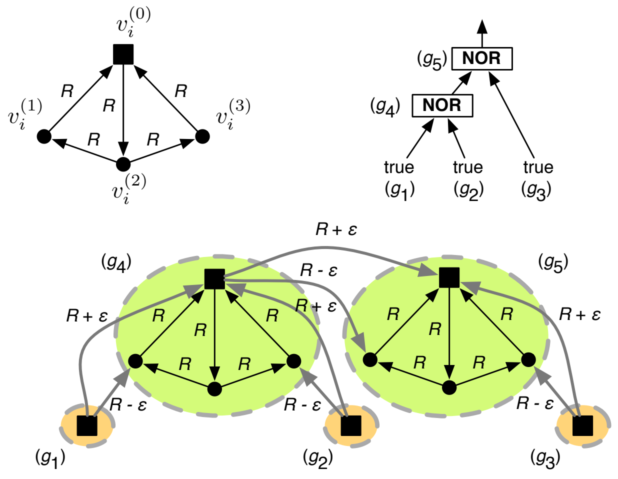


Figure 5.2: Top (left): the gadget used in the find-and-halve reduction. Top (right): an example of a circuit. Bottom: the graph produced by performing the reduction on the circuit from the middle figure.

Analysis: We reason about the reduction as follows. First, notice that this reduction can be accomplished in $\widetilde{O}(1)$ depth. Furthermore, the gadget has the property that any $v_i^{(0)}$ (square) cannot become a CCV unless both $v_i^{(1)}$ (left arm) and $v_i^{(3)}$ (right arm) are removed (because other nodes are chosen as CCVs and they can cover these nodes with $2R$ from them). That is, for a square node to be selected as a CCV, both the left and right arms have to be "unlocked."

The rough intuition behind this reduction is that find-and-halve is forced to process vertices in layers (topological ordering of the gates in the circuit) parallel to how circuit evaluation is carried out. Consider the asymmetric k-center instance derived from the reduction. Initially, only the input nodes are CCVs and by definition of NOR-CVP, these nodes represent `true`. If a vertex is CCV, either it is an input node or both of its arms have been unlocked. Furthermore, when a node $v_i^{(0)}$ is chosen as a CCV, it wipes out every node reachable within $2R$, eliminating the square nodes of the gadgets that consume the output of g_i, together with an arm of the gadget one-level further up. This corresponding to the following scenario in the circuit: if a gate g_i outputs a `true`, the gate that consumes the output of g_i has to be `false`—and the gates one-level further up have one input set to `false` and have a potential to output `true`. Therefore, it is easily verified that **(1)** if any input to a gate g_i is `true`, then $v_i^{(0)}$, together with the arm that takes that input, will be removed and can never become CCV; and **(2)** if the input to a g_i gate is `false`, the arm corresponding to this input will be removed (i.e., covered by some prior CCV down the line). Arguing inductively, we have the theorem.

5.5 Greedy Facility Location

Metric facility location is a fundamental problem in approximation algorithms; detailed problem description, as well as bibliographical remarks, appears in Chapter 3. To review, the input consists of a set of *facilities* F and a set of *clients* C, where each facility $i \in F$ has cost f_i, and each client $j \in C$ incurs $d(j, i)$ to use facility i—and the goal is to find a set of facilities $F_S \subseteq F$ that minimizes the objective function $\textsc{FacLoc}(F_S) = \sum_{i \in F_S} f_i + \sum_{j \in C} d(j, F_S)$. The distance d is assumed to be symmetric and satisfy the triangle inequality. This problem has an exceptionally simple factor-1.861 greedy algorithm due to Jain et al. [JMM+03], which has been parallelized by Blelloch and Tangwongsan [BT10], yielding an RNC $(3.722+\varepsilon)$-approximation with work $O(p \log^2 p)$, where $p = |F| \times |C|$ (see Chapter 3).

Using ideas from previous sections, we develop an RNC algorithm with an improved approximation guarantee, which essentially matches that of the sequential version.

Theorem 5.19 *Let $\varepsilon > 0$ be a small constant. There is a $O(p \log p)$-work $O(\log^4 p)$-depth factor-$(1.861 + \varepsilon)$ approximation algorithm for the (metric) facility location problem.*

Recall the definitions of a star, price, and a maximal star from Definition 3.2. This will be essential in describing the algorithm.

Presented in Algorithm 5.5.1 is a parallel greedy approximation algorithm for metric facility location. The algorithm closely mimics the behaviors of Jain et al.'s algorithm, except the parallel algorithm is more aggressive in choosing the stars to open. Consider the natural integer-program formulation of facility location for which the relaxation yields the pair of primal and dual programs shown in Figure 3.1 (Chapter 3). We wish to give a dual-fitting analysis similar to that of Jain et al.

Algorithm 5.5.1 Parallel greedy algorithm for metric facility location.

```
Set $F_A = \emptyset$. For $t = 1, 2, \ldots,$ until $C = \emptyset$,
  i. Let $\tau^{(t)} = (1+\varepsilon)\cdot \min\{\mathsf{best}(i) : i \in F\}$ and $F^{(t)} = \{i \in F : \mathsf{best}(i) \le \tau^{(t)}\}$.
  ii. Build a bipartite graph $G$ from $F^{(t)}$ and $N(\cdot)$, where for each $i \in F^{(t)}$, $N(i) = \{j \in C : d(j,i) \le \tau^{(t)}\}$.
  iii. While $(F^{(t)} \neq \emptyset)$, repeat:
      1. Compute $\Delta J^{(t)} = \mathtt{MaNIS}_{(\varepsilon,3\varepsilon)}(G)$ and $J^{(t)} = J^{(t)} \cup \Delta J^{(t)}$
      2. Remove $\Delta J^{(t)}$ and $N(\Delta J^{(t)})$ from $G$, remove the clients $N(\Delta J^{(t)})$ from $C$, and set $f_i = 0$ for all $i \in \Delta J^{(t)}$.
      3. Delete any $i \in F^{(t)}$ such that $\mathsf{price}((i, N(i))) > \tau^{(t)}$.
```

Their proof shows that the solution's cost is equal to the sum of α_j's over the clients, where α_j is the price of the star with which client j is connected up. Following this analysis, we set α_j to $\tau^{(t)}/(1+\varepsilon)$ where t is the iteration that the client was removed. Note that stars chosen in $F^{(t)}$ may have overlapping clients. For this reason, we cannot afford to open them all, or we would not be able to bound the solution's cost by the sum of α_j's. This situation, however, is rectified by the use of MaNIS, allowing us to prove Lemma 5.22, which relates the cost of the solution to α_j's. Before we prove this lemma, two easy-to-check facts are in order:

Fact 5.20 *In the graph G constructed in Step 2, for all $i \in F^{(t)}$,* $\mathsf{price}((i, N(i))) \le \tau^{(t)}$.

Fact 5.21 *At any point during iteration t,* $\mathsf{best}(i) \le \tau^{(t)}$ *if and only if* $\mathsf{price}((i, N(i))) \le \tau^{(t)}$.

Lemma 5.22 *The cost of the algorithm's solution* $\textsc{FacLoc}(F_A)$ *is upper-bounded by* $\frac{1}{1-5\varepsilon}\sum_{j\in C}\alpha_j$.

Proof: Let $t > 0$ and consider what happens inside the inner loop (Steps iii.1—iii.3). Each iteration of the inner loop runs MaNIS on $F^{(t)}$ with each $i \in F^{(t)}$ satisfying $\tau^{(t)} \ge \mathsf{price}((i, N(i))) = (f_i + \sum_{j\in N(i)} d(j,i))/|N(i)|$, so

$$|N(i)| \ge \tfrac{1}{\tau^{(t)}}\Big(f_i + \textstyle\sum_{j\in N(i)} d(j,i)\Big).$$

Because of Fact 5.20 and Step iii.3, the relationship $\tau^{(t)} \ge \mathsf{price}((i, N(i)))$ is maintained throughout iteration t. Running a $(\varepsilon, 3\varepsilon)$-MaNIS on $F^{(t)}$ ensures that each $\Delta J^{(t)}$ satisfies

$|N(\Delta J^{(t)})| \geq (1-4\varepsilon)\sum_{i\in\Delta J^{(t)}}|N(i)|$. Thus,

$$\sum_{j\in\Delta J^{(t)}} \alpha_j = \frac{\tau^{(t)}}{1+\varepsilon}|N(\Delta J^{(t)})| \geq \frac{\tau^{(t)}}{1+\varepsilon}(1-4\varepsilon)\sum_{i\in\Delta J^{(t)}}|N(i)|$$
$$\geq (1-5\varepsilon)\sum_{i\in\Delta J^{(t)}}\Big(f_i + \sum_{j\in N(i)} d(j,i)\Big),$$

which is at least

$$(1-5\varepsilon)\Big(\sum_{i\in\Delta J^{(t)}} f_i + \sum_{j\in N(\Delta J^{(t)})} d(j,\Delta J^{(t)})\Big).$$

Since every client has to appear in at least one $J^{(t)}$, summing across the inner loop's iterations and t gives the lemma. ■

In the series of claims that follows, we show that when scaled down by a factor of $\gamma = 1.861$, the α setting determined above is a dual feasible solution. We will assume without loss of generality that $\alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_{|C|}$. Let $W_i = \{j \in C : \alpha_j \geq \gamma \cdot d(j,i)\}$ for all $i \in F$ and $W = \cup_i W_i$.

Claim 5.23 *For any facility $i \in F$ and client $j_0 \in C$,*

$$\textstyle\sum_{j\in W: j\geq j_0} \max(0, \alpha_{j_0} - d(j,i)) \quad\leq\quad f_i.$$

Proof: Suppose for a contradiction that there exist client j and facility i such that the inequality in the claim does not hold. That is,

$$\sum_{j\in W: j\geq j_0} \max(0, \alpha_{j_0} - d(j,i)) \quad>\quad f_i \tag{5.1}$$

Let t be the iteration such that $\alpha_{j_0} = \tau^{(t)}/(1+\varepsilon)$. Let $\widehat{C}^{(t)}$ be the set of clients j's such that $\alpha_{j_0} - d(j,i) > 0$ that remain at the beginning of iteration t. Thus, by our assumption and the fact that $\{j \in W : j \geq j_0 \wedge \alpha_{j_0} > d(j,i)\} \subseteq \widehat{C}^{(t)}$, we establish $\sum_{j\in\widehat{C}^{(t)}} \alpha_{j_0} - d(j,i) = \sum_{j\in\widehat{C}^{(t)}} \max(0,\alpha_{j_0} - d(j,i)) \geq \sum_{j\in W: j\geq j_0} \max(0, \alpha_{j_0} - d(j,i)) > f_i$. It follows that $\alpha_{j_0} > \frac{1}{|\widehat{C}^{(t)}|}(f_i + \sum_{j\in\widehat{C}^{(t)}} d(j,i))$. Hence, $\tau^{(t)}/(1+\varepsilon) = \alpha_{j_0} > \frac{1}{|\widehat{C}^{(t)}|}(f_i + \sum_{j\in\widehat{C}^{(t)}} d(j,i)) \geq \mathsf{best}(i) \geq \tau^{(t)}/(1+\varepsilon)$ since $\tau^{(t)}/(1+\varepsilon)$ is the minimum of the best price in that iteration. This gives a contradiction, proving the claim. ■

Claim 5.24 *Let $i \in F$, and $j, j' \in W$ be clients. Then, $\alpha_j \leq \alpha_{j'} + d(i,j') + d(i,j)$.*

Proof: If $\alpha_j \leq \alpha_{j'}$, the proof is trivial, so assume $\alpha_j > \alpha_{j'}$. Let i' be any facility that removed j' (i.e, $i' \in \Delta J^{(t)}$ such that $j \in N(i')$). It suffices to show that $\alpha_j \leq d(i', j)$ and the claim follows from triangle inequality. Since $\alpha_j > \alpha_{j'}$, in the iteration t where $\alpha_j = \tau^{(t)}/(1+\varepsilon)$, we know that $f_{i'}$ has already been set to 0, so $\mathsf{best}(i') \leq d(j, i')$. Furthermore, in this iteration, $\alpha_j \leq \mathsf{best}(i')$ as $\alpha_j = \min\{\mathsf{best}(i)\}$, proving the claim. ■

These two claims are sufficient to set up a factor-revealing LP identical to Jain et al.'s. Therefore, the following lemma follows from Jain et al. [JMM$^+$03] (Lemmas 3.4 and 3.6):

Lemma 5.25 *The setting $\alpha'_j = \frac{\alpha_j}{\gamma}$ and $\beta'_{ij} = \max(0, \alpha'_j - d(j,i))$ is a dual feasible solution, where $\gamma = 1.861$.*

Combining this lemma with Lemma 5.22 and weak duality, we have the promised approximation guarantee.

Running time analysis: Fix $\varepsilon > 0$. We argue that the number of rounds is upper bounded by $O(\log p)$. For this, we need a preprocessing step which ensures that the ratio between the largest τ and the smallest τ ever encountered in the algorithm is $O(p^{O(1)})$. This can be done in $O(p)$ work and $O(\log(|F|+|C|))$ depth, adding $\varepsilon \cdot \mathsf{opt}$ to the solution's cost [BT10]. Armed with that, it suffices to show the following claim.

Claim 5.26 $\tau^{(t+1)} \geq (1+\varepsilon) \cdot \tau^{(t)}$.

Proof: Let $\mathsf{best}^{(t)}(i)$ denote $\mathsf{best}(i)$ at the beginning of iteration t. Let i^* be the facility whose $\mathsf{best}^{(t+1)}(i^*)$ attains $\tau^{(t+1)}/(1+\varepsilon)$. To prove the claim, it suffices to show that $\mathsf{best}^{(t+1)}(i^*) \geq \tau^{(t)}$, as this will imply $\tau^{(t+1)} \geq (1+\varepsilon) \cdot \tau^{(t)}$. Now consider two possibilities.

— *Case 1.* i^* was part of $F^{(t)}$, so then either i^* was opened in this iteration or i^* was removed from $F^{(t)}$ in Step 3.iii. If i^* was opened, all clients at distance at most $\tau^{(t)}$ from it would be connected up, so $\mathsf{best}^{(t+1)}(i^*) \geq \tau^{(t)}$. Otherwise, i^* was removed in Step iii.3, in which case $\mathsf{best}^{(t+1)}(i^*) \geq \tau^{(t)}$ by the removal criteria and Fact 5.21.

— *Case 2.* Otherwise, i^* was not part of $F^{(t)}$. This means that $\mathsf{best}^{(t)}(i^*) > \tau^{(t)}$. As the set of unconnected clients can only become smaller, the price of the best star centered at i^* can only go up. So $\mathsf{best}^{(t+1)}(i^*)$ will be at least $\tau^{(t)}$, which in turn implies the claim. ■

Thus, the total number of iterations (outer loop) in the algorithm is $O(\log p)$. We now consider the work and depth of each iteration. Step 1 involves computing $\mathsf{best}(i)$ for all $i \in F$. This can be done in $O(p)$ work and $O(\log p)$ depth using a prefix computation and standard techniques (see [BT10] for details). Step 2 can be done in the same work-depth bounds. Inside the inner loop, each MaNIS call requires $O(p')$ work and $O(\log^2 p')$ depth, where m' is the number of edges in G. Steps iii.2–3 do not require more than $O(p')$ work and $O(\log p')$ depth. Furthermore, note that if $i \in F^{(t)}$ is not chosen by MaNIS, it loses at least an ε fraction of its neighbors. Therefore, the total work of in the inner loop (for each t) is $O(\varepsilon^{-1}p)$, and depth $O(\log^3 p)$. Combining these gives the theorem.

5.6 Conclusion

We formulated and studied MaNIS—a graph abstraction of a problem at the crux of many (set) covering-type problem. We gave a linear-work RNC solution to this problem and applied it to derive parallel approximation algorithms for several problems, yielding RNC algorithms for set cover, (prefix-optimal) max cover, min-sum set cover, asymmetric k-center, and metric facility location.

Chapter 6

Parallel Low-Stretch Spanning Subtrees and Subgraphs, and Parallel SDD Solvers

6.1 Background and Motivations

Solving a system of linear equations $Ax = b$ is a fundamental computing primitive that lies at the core of many numerical and scientific computing algorithms, including the popular interior-point algorithms. The special case of symmetric diagonally dominant (SDD) systems has seen substantial progress in recent years; in particular, the ground-breaking work of Spielman and Teng showed how to solve SDD systems to accuracy ε in time $\widetilde{O}(m \log(1/\varepsilon))$, where m is the number of non-zeros in the $n \times n$-matrix A.[1] This is algorithmically significant since solving SDD systems has implications to computing eigenvectors, solving flow problems, finding graph sparsifiers, and problems in vision and graphics (see [Spi10, Ten10] for these and other applications).

In the sequential setting, the current best SDD solvers run in $O(m \log n (\log\log n)^2 \log(\frac{1}{\varepsilon}))$ time [KMP11]. However, with the exception of the special case of planar SDD systems [KM07],

[1]The Spielman-Teng solver and all subsequent improvements are randomized algorithms. As a consequence, all algorithms relying on the solvers are also randomized. For simplicity, we omit standard complexity factors related to the probability of error.

we know of no previous parallel SDD solvers that perform near-linear[2] work and achieve non-trivial parallelism. This raises a natural question: *Is it possible to solve an SDD linear system in $o(n)$ depth and $\widetilde{O}(m)$ work?* We answer this question affirmatively:

Theorem 6.1 *For any fixed $\theta > 0$ and any $\epsilon > 0$, there is an algorithm `SDDSolve` that on input an $n \times n$ SDD matrix A with m non-zero elements and a vector b, computes a vector $\tilde{x}$ such that $\|\tilde{x} - A^{+}b\|_A \le \varepsilon \cdot \|A^{+}b\|_A$ in $O(m \log^{O(1)} n \log \frac{1}{\epsilon})$ work and $O(m^{1/3+\theta} \log \frac{1}{\epsilon})$ depth.*

In the process, we give parallel algorithms for constructing graph decompositions with strong-diameter guarantees, and parallel algorithms to construct low-stretch spanning trees and low-stretch ultra-sparse subgraphs, which may be of independent interest. An overview of these algorithms and their underlying techniques is given in Section 6.3.

Some Applications. Let us mention some of the implications of Theorem 6.1, obtained by plugging it into known reductions.

— *Construction of (Spectral) Sparsifiers.* Spielman and Srivastava [SS08] showed that spectral sparsifiers can be constructed using $O(\log n)$ Laplacian solves, and using our theorem we get spectral and cut sparsifiers in $\tilde{O}(m^{1/3+\theta})$ depth and $\tilde{O}(m)$ work.

— *Flow Problems.* Daitsch and Spielman [DS08] showed that various graph optimization problems, such as max-flow, min-cost flow, and lossy flow problems, can be reduced to $\widetilde{O}(m^{1/2})$ applications[3] of SDD solves via interior point methods described in [Ye97, Ren01, BV04b]. Combining this with our main theorem implies that these algorithms can be parallelized to run in $\widetilde{O}(m^{5/6+\theta})$ depth and $\widetilde{O}(m^{3/2})$ work. This gives the first parallel algorithm with $o(n)$ depth which is work-efficient to within polylog(n) factors relative to the sequential algorithm for all problems analyzed in [DS08]. Furthermore, our results in conjunction of a recent result of Christiano et al. [CKM+11] give a $\widetilde{O}(\varepsilon^{-O(1)}m^{4/3})$-work, $\widetilde{O}(\varepsilon^{-O(1)}m^{2/3})$-depth algorithm for $(1-\varepsilon)$ approximate max-flow and $(1+\varepsilon)$ min-cut.

In some sense, the parallel bounds are more interesting than the sequential times because though in many cases the results in [DS08] are not the best known sequentially (e.g. max-flow), they do lead to the best know parallel bounds for problems that have traditionally been hard to parallelize. Finally, we note that although [DS08] does not explicitly analyze shortest path, their analysis naturally generalizes the LP for it.

[2]i.e. linear up to polylog factors.

[3]here $\tilde{O}$ hides $\log U$ factors as well, where it's assumed that the edge weights are integers in the range $[1 \ldots U]$

6.2 Preliminaries and Notation

Given a graph $G = (V, E)$, let *hop*(u, v) denote the *edge-count distance* (or hop distance) between u and v, ignoring the edge lengths. When the graph has edge lengths $w(e)$ (also denoted by w_e), let $d_G(u, v)$ denote the *edge-length distance*, the shortest path (according to these edge lengths) between u and v. If the graph has unit edge lengths, the two definitions coincide. We drop subscripts when the context is clear. We denote by $V(G)$ and $E(G)$, respectively, the set of nodes and the set of edges, and use $n = |V(G)|$ and $m = |E(G)|$. For an edge $e = \{u, v\}$, the stretch of e on G' is $\text{str}_{G'}(e) = d_{G'}(u, v)/w(e)$. The *total stretch* of $G = (V, E, w)$ with respect to G' is $\text{str}_{G'}(E(G)) = \sum_{e \in E(G)} \text{str}_{G'}(e)$.

Given $G = (V, E)$, a distance function δ (which is either *hop* or d), and a partition of V into $C_1 \uplus C_2 \uplus \ldots \uplus C_p$, let $G[C_i]$ denote the induced subgraph on set C_i. Here, the operator $\uplus$ denotes a disjoint union. The *weak diameter* of C_i is $\max_{u,v \in C_i} \delta_G(u, v)$, whereas the *strong diameter* of C_i is $\max_{u,v \in C_i} \delta_{G[C_i]}(u, v)$; the former measures distances in the original graph, whereas the latter measures distances within the induced subgraph. The strong (or weak) diameter of the partition is the maximum strong (or weak) diameter over all the components C_i's.

Graph Laplacians. For a fixed, but arbitrary, numbering of the nodes and edges in a graph $G = (V, E)$, the Laplacian L_G of G is the $|V|$-by-$|V|$ matrix given by

$$L_G(i, j) = \begin{cases} -w_{ij} & \text{if } i \neq j \\ \sum_{\{j,i\} \in E(G)} w_{ij} & \text{if } i = j \end{cases},$$

When the context is clear, we use G and L_G interchangeably. Given two graphs G and H and a scalar $\mu \in \mathbb{R}$, we say $G \preceq \mu H$ if $\mu L_H - L_G$ is positive semidefinite, or equivalently $x^\top L_G x \leq \mu x^\top L_H x$ for all vector $x \in \mathbb{R}^{|V|}$.

Matrix Norms, SDD Matrices. For a matrix A, we denote by A^+ the Moore-Penrose pseudoinverse of A (i.e., A^+ has the same null space as A and acts as the inverse of A on its image). Given a symmetric positive semi-definite matrix A, the A-*norm* of a vector x is defined as $\|x\|_A = \sqrt{x^\top A x}$. A matrix A is symmetrically diagonally dominant (SDD) if it is symmetric and for all i, $A_{i,i} \geq \sum_{j \neq i} |A_{i,j}|$. Solving an SDD system reduces in $O(m)$ work and $O(\log^{O(1)} m)$ depth to solving a graph Laplacian (a subclass of SDD matrices corresponding to undirected weighted graphs) [Gre96, Section 7.1].

Parallel Ball Growing. Let $B_G(s, r)$ denote the ball of edge-count distance r from a source s, i.e., $B_G(s, r) = \{v \in V(G) : hop_G(s, v) \leq r\}$. We rely on an elementary form of parallel breadth-first search to compute $B_G(s, r)$. The algorithm visits the nodes level by level as they are encountered in the BFS order. More precisely, level 0 contains only the source node s, level 1 contains the neighbors of s, and each subsequent level $i + 1$ contains the neighbors of level i's nodes that have not shown up in a previous level. On standard parallel models (e.g., CRCW), this can be computed in $O(r \log n)$ depth and $O(m' + n')$ work, where m' and n' are the total numbers of edges and nodes, respectively, encountered in the search [UY91, KS97]. Notice that we could achieve this runtime bound with a variety of graph (matrix) representations, e.g., using the compressed sparse-row (CSR) format. Our applications apply ball growing on r roughly $O(\log^{O(1)} n)$, resulting in a small depth bound. We remark that the idea of small-radius parallel ball growing has previously been employed in the context of approximate shortest paths (see, e.g., [UY91, KS97, Coh00]). There is an alternative approach of repeatedly squaring a matrix, which gives a better depth bound for large r *at the expense* of a much larger work bound (about n^3).

Finally, we state a tail bound which will be useful in our analysis. This bound is easily derived from well-known facts about the tail of a hypergeometric random variable [Chv79b, Hoe63, Ska09].

Lemma 6.2 (Hypergeometric Tail Bound) *Let H be a hypergeometric random variable denoting the number of red balls found in sample of n drawn from a total of N balls of which M are red. Then, if $\mu = \mathbf{E}[H] = nM/N$, then*

$$\mathbf{Pr}[H \geq 2\mu] \quad \leq \quad e^{-\mu/4}$$

Proof: We apply the following theorem of Hoeffding [Chv79b, Hoe63, Ska09]. For any $t > 0$,

$$\mathbf{Pr}[H \geq \mu + tn] \leq \left(\left(\frac{p}{p+t}\right)^{p+t} \left(\frac{1-p}{1-p-t}\right)^{1-p-t} \right)^n,$$

where $p = \mu/n$. Using $t = p$, we have

$$\begin{aligned}
\mathbf{Pr}[H \geq 2\mu] &\leq \left(\left(\frac{p}{2p}\right)^{2p} \left(\frac{1-p}{1-2p}\right)^{1-2p} \right)^n \\
&\leq \left(e^{-p\ln 4} \left(1 + \frac{p}{1-2p}\right)^{1-2p} \right)^n \\
&\leq \left(e^{-p\ln 4} \cdot e^p \right)^n \\
&\leq e^{-\frac{1}{4}pn},
\end{aligned}$$

where we have used the fact that $1 + x \leq \exp(x)$. ■

6.3 Overview of Our Techniques

In the general solver framework of Spielman and Teng [ST06, KMP10], near linear-time SDD solvers rely on a suitable preconditioning chain of progressively smaller graphs. Assuming that we have an algorithm for generating low-stretch spanning trees, the algorithm as given in [KMP10] parallelizes under the following modifications: (i) perform the partial Cholesky factorization in parallel and (ii) terminate the preconditioning chain with a graph that is of size approximately $m^{1/3}$. The details in Section 6.6 are the primary motivation of the main technical part of the work in this chapter, a parallel implementation of a modified version of Alon et al.'s low-stretch spanning tree algorithm [AKPW95].

More specifically, as a first step, we find an embedding of graphs into a spanning tree with average stretch $2^{O(\sqrt{\log n \log\log n})}$ in $\widetilde{O}(m)$ work and $O(2^{O(\sqrt{\log n \log\log n})} \log \Delta)$ depth, where Δ is the ratio of the largest to smallest distance in the graph. The original AKPW algorithm relies on a parallel graph decomposition scheme of Awerbuch [Awe85], which takes an unweighted graph and breaks it into components with a specified diameter and few crossing edges. While such schemes are known in the sequential setting, they do not parallelize readily because removing edges belonging to one component might increase the diameter or even disconnect subsequent components. We present the first near linear-work parallel decomposition algorithm that also gives strong-diameter guarantees, in Section 6.4, and the tree embedding results in Section 6.5.1.

Ideally, we would have liked for our spanning trees to have a polylogarithmic stretch, computable by a polylogarithmic depth, near linear-work algorithm. However, for our solvers, we make the additional observation that we do not really need a spanning *tree* with small stretch; it suffices to give an "ultra-sparse" graph with small stretch, one that has only $O(m/\operatorname{polylog}(n))$ edges more than a tree. Hence, we present a parallel algorithm in Section 6.5.2 which outputs an ultra-sparse graph with $O(\operatorname{polylog}(n))$ average stretch, performing $\widetilde{O}(m)$ work with $O(\operatorname{polylog}(n))$ depth. Note that this removes the dependence of $\log \Delta$ in the depth, and reduces both the stretch and the depth from $2^{O(\sqrt{\log n \log\log n})}$ to $O(\operatorname{polylog}(n))$.[4] When combined with the aforementioned routines for constructing a SDD solver presented in Section 6.6, this low-stretch spanning subgraph construction yields a parallel solver algorithm.

[4]As an aside, this construction of low-stretch ultra-sparse graphs shows how to obtain the $\widetilde{O}(m)$-time linear system solver of Spielman and Teng [ST06] without using their low-stretch spanning trees result [EEST05, ABN08].

6.4 Parallel Low-Diameter Decomposition

In this section, we present a parallel algorithm for partitioning a graph into components with low (strong) diameter while cutting only a few edges in each of the k disjoint subsets of the input edges. The sequential version of this algorithm is at the heart of the low-stretch spanning tree algorithm of Alon, Karp, Peleg, and West (AKPW) [AKPW95].

For context, notice that the outer layer of the AKPW algorithm (more details in Section 6.5) can be viewed as bucketing the input edges by weight, then partitioning and contracting them repeatedly. In this view, a number of edge classes are "reduced" simultaneously in an iteration. Further, as we wish to output a spanning subtree at the end, the components need to have low strong-diameter (i.e., one could not take "shortcuts" through other components). In the sequential case, the strong-diameter property is met by removing components one after another, but this process does not parallelize readily. For the parallel case, we guarantee this by growing balls from multiple sites, with appropriate "jitters" that conceptually delay when these ball-growing processes start, and assigning vertices to the first region that reaches them. These "jitters" terms are crucial in controlling the probability that an edge goes across regions. But this probability also depends on the number of regions that could reach such an edge. To keep this number small, we use a repeated sampling procedure motivated by Cohen's (β, W)-cover construction [Coh93].

More concretely, we prove the following theorem:

Theorem 6.3 (Parallel Low-Diameter Decomposition) *Given an input graph $G = (V, E_1 \uplus \ldots \uplus E_k)$ with k edge classes and a "radius" parameter ρ, the algorithm $\texttt{Partition}(G, \rho)$, upon termination, outputs a partition of V into components $\mathcal{C} = (C_1, C_2, \ldots, C_p)$, each with center s_i such that*

1. *the center $s_i \in C_i$ for all $i \in [p]$,*
2. *for each i, every $u \in C_i$ satisfies $\mathrm{hop}_{G[C_i]}(s_i, u) \le \rho$, and*
3. *for all $j = 1, \ldots, k$, the number of edges in E_j that go between components is at most $|E_j| \cdot \frac{c_1 \cdot k \log^3 n}{\rho}$, where c_1 is an absolute constant.*

Furthermore, $\texttt{Partition}$ runs in $O(m \log^2 n)$ expected work and $O(\rho \log^2 n)$ expected depth.

Low-Diameter Decomposition for Simple Unweighted Graphs

To prove this theorem, we begin by presenting an algorithm `splitGraph` that works with simple graphs with only *one* edge class and describe how to build on top of it an algorithm that handles multiple edge classes.

The basic algorithm takes as input a simple, unweighted graph $G = (V, E)$ and a radius (in hop count) parameter ρ and outputs a partition V into components $C_1, \ldots, C_p$, each with center s_i, such that

(P1) Each center belongs to its own component. That is, the center $s_i \in C_i$ for all $i \in [p]$;
(P2) Every component has radius at most ρ. That is, for each $i \in [p]$, every $u \in C_i$ satisfies $\mathit{hop}_{G[C_i]}(s_i, u) \leq \rho$;
(P3) Given a technical condition (to be specified) that holds with probability at least $3/4$, the probability that an edge of the graph G goes between components is at most $\frac{136}{\rho} \log^3 n$.

In addition, this algorithm runs in $O(m \log^2 n)$ expected work and $O(\rho \log^2 n)$ expected depth. (These properties should be compared with the guarantees in Theorem 6.3.)

Consider the pseudocode of this basic algorithm in Algorithm 6.4.1. The algorithm takes as input an unweighted n-node graph G and proceeds in $T = O(\log n)$ iterations, with the eventual goal of outputting a partition of the graph G into a collection of sets of nodes (each set of nodes is known as a component). Let $G^{(t)} = (V^{(t)}, E^{(t)})$ denote the graph at the beginning of iteration t. Since this graph is unweighted, the distance in this algorithm is always the hop-count distance $\mathit{hop}(\cdot, \cdot)$. For iteration $t = 1, \ldots, T$, the algorithm picks a set of starting centers $S^{(t)}$ to grow balls from; as with Cohen's (β, W)-cover, the number of centers is progressively larger with iterations, reminiscent of the doubling trick (though with more careful handling of the growth rate), to compensate for the balls' shrinking radius and to ensure that the graph is fully covered.

Still within iteration t, it chooses a random "jitter" value $\delta_s^{(t)} \in_R \{0, 1, \ldots, R\}$ for each of the centers in $S^{(t)}$ and grows a ball from each center s out to radius $r^{(t)} - \delta_s^{(t)}$, where $r^{(t)} = \frac{\rho}{2 \log n}(T - t + 1)$. Let $X^{(t)}$ be the union of these balls (i.e., the nodes "seen" from these starting points). In this process, the "jitter" should be thought of as a random amount by which we delay the ball-growing process on each center, so that we could assign nodes to the first region that reaches them while being in control of the number of cross-component edges. Equivalently, our algorithm forms the components by assigning each vertex u reachable from one of these centers to the center that minimizes $\mathit{hop}_{G^{(t)}}(u, s) + \delta_s^{(t)}$ (ties broken in a consistent manner, e.g., lexicographically). Note that because of these "jitters," some centers might not be assigned any vertex, not even itself. For centers that are assigned some nodes, we include their components in the output, designating them as the components' centers. Finally, we construct $G^{(t+1)}$ by removing nodes that were "seen" in this iteration (i.e., the nodes in $X^{(t)}$)—because they are already part of one of the output components—and adjusting the edge set accordingly.

Algorithm 6.4.1 `splitGraph` $(G = (V, E), \rho)$ — Split an input graph $G = (V, E)$ into components of hop-radius at most ρ.

```
Let $G^{(1)} = (V^{(1)}, E^{(1)}) \leftarrow G$. Define $R = \rho/(2\log n)$. Create empty collection of components $\mathcal{C}$.
Use $hop^{(t)}$ as shorthand for $hop_{G^{(t)}}$, and define $B^{(t)}(u, r) \overset{\text{def}}{=} B_{G^{(t)}}(u, r) = \{v \in V^{(t)} \mid hop^{(t)}(u, v) \leq r\}$.
For $t = 1, 2, \ldots, T = 2\log_2 n$,
    1. Randomly sample $S^{(t)} \subseteq V^{(t)}$, where $|S^{(t)}| = \sigma_t = 12n^{t/T-1}|V^{(t)}|\log n$, or use $S^{(t)} = V^{(t)}$ if $|V^{(t)}| < \sigma_t$.
    2. For each "center" $s \in S^{(t)}$, draw $\delta_s^{(t)}$ uniformly at random from $\mathbb{Z} \cap [0, R]$.
    3. Let $r^{(t)} \leftarrow (T - t + 1)R$.
    4. For each center $s \in S^{(t)}$, compute the ball $B_s^{(t)} = B^{(t)}(s, r^{(t)} - \delta_s^{(t)})$.
    5. Let $X^{(t)} = \cup_{s \in S^{(t)}} B_s^{(t)}$.
    6. Create components $\{C_s^{(t)} \mid s \in S^{(t)}\}$ by assigning each $u \in X^{(t)}$ to the component $C_s^{(t)}$ such that $s$ minimizes $hop_{G^{(t)}}(u, s) + \delta_s^{(t)}$ (breaking ties lexicographically).
    7. Add non-empty $C_s^{(t)}$ components to $\mathcal{C}$.
    8. Set $V^{(t+1)} \leftarrow V^{(t)} \setminus X^{(t)}$, and let $G^{(t+1)} \leftarrow G^{(t)}[V^{(t+1)}]$. Quit early if $V^{(t+1)}$ is empty.
Return $\mathcal{C}$.
```

Analysis. Throughout this analysis, we make reference to various quantities in the algorithm and assume the reader's basic familiarity with our algorithm. We begin by proving properties (P1)–(P2). First, we state an easy-to-verify fact, which follows immediately by our choice of radius and components' centers.

Fact 6.4 *If vertex u lies in component $C_s^{(t)}$, then $hop^{(t)}(s, u) \leq r^{(t)}$. Moreover, $u \in B_s^{(t)}$.*

We also need the following lemma to argue about strong diameter.

Lemma 6.5 *If vertex $u \in C_s^{(t)}$, and vertex $v \in V^{(t)}$ lies on any u-s shortest path in $G^{(t)}$, then $v \in C_s^{(t)}$.*

Proof: Since $u \in C_s^{(t)}$, Fact 6.4 implies u belongs to $B_s^{(t)}$. But $hop^{(t)}(v, i) < hop^{(t)}(u, i)$, and hence v belongs to $B_s^{(t)}$ and $X^{(t)}$ as well. This implies that v is assigned to *some* component $C_j^{(t)}$; we claim $j = s$.

For a contradiction, assume that $j \neq s$, and hence $hop^{(t)}(v, j) + \delta_j^{(t)} \leq hop^{(t)}(v, s) + \delta_s^{(t)}$. In this case $hop^{(t)}(u, j) + \delta_j^{(t)} \leq hop^{(t)}(u, v) + hop^{(t)}(v, j) + \delta_j^{(t)}$ (by the triangle inequality). Now using the assumption, this expression is at most $hop^{(t)}(u, v) + hop^{(t)}(v, s) + \delta_s^{(t)} =$

$hop^{(t)}(u,s) + \delta_s^{(t)}$ (since v lies on the shortest u-s path). But then, u would be also assigned to $C_j^{(t)}$, a contradiction. ■

Hence, for each non-empty component $C_s^{(t)}$, its center s lies within the component (since it lies on the shortest path from s to any $u \in C_s^{(t)}$), which proves (P1). Moreover, by Fact 6.4 and Lemma 6.5, the (strong) radius is at most TR, proving (P2). It now remains to prove (P3), and the work and depth bounds.

Lemma 6.6 *For any vertex $u \in V$, with probability at least $1 - n^{-6}$, there are at most $68 \log^2 n$ pairs*[5] *(s,t) such that $s \in S^{(t)}$ and $u \in B^{(t)}(s, r^{(t)})$,*

We will prove this lemma in a series of claims.

Claim 6.7 *For $t \in [T]$ and $v \in V^{(t)}$, if $|B^{(t)}(v, r^{(t+1)})| \geq n^{1-t/T}$, then $v \in X^{(t)}$ w.p. at least $1 - n^{-12}$.*

Proof: First, note that for any $s \in S^{(t)}$, $r^{(t)} - \delta_s \geq r^{(t)} - R = r^{(t+1)}$, and so if $s \in B^{(t)}(v, r^{(t+1)})$, then $v \in B_s^{(t)}$ and hence in $X^{(t)}$. Therefore,

$$\textbf{Pr}\left[v \in X^{(t)}\right] \geq \textbf{Pr}\left[S^{(t)} \cap B^{(t)}(v, r^{(t+1)}) \neq \emptyset\right],$$

which is the probability that a random subset of $V^{(t)}$ of size σ_t hits the ball $B^{(t)}(v, r^{(t+1)})$. But, $\textbf{Pr}\left[S^{(t)} \cap B^{(t)}(v, r^{(t+1)}) \neq \emptyset\right] \geq 1 - \left(1 - \frac{|B^{(t)}(v, r^{(t+1)})|}{|V^{(t)}|}\right)^{\sigma_t}$, which is at least $1 - n^{-12}$. ■

Claim 6.8 *For $t \in [T]$ and $v \in V$, the number of $s \in S^{(t)}$ such that $v \in B^{(t)}(s, r^{(t)})$ is at most $34 \log n$ w.p. at least $1 - n^{-8}$.*

Proof: For $t = 1$, the size $\sigma_1 = O(\log n)$ and hence the claim follows trivially. For $t \geq 2$, we condition on all the choices made in rounds $1, 2, \ldots, t-2$. Note that if v does not survive in $V^{(t-1)}$, then it does not belong to $V^{(t)}$ either, and the claim is immediate. So, consider two cases, depending on the size of the ball $B^{(t-1)}(v, r^{(t)})$ in iteration $t-1$:

— *Case 1.* If $|B^{(t-1)}(v, r^{(t)})| \geq n^{1-(t-1)/T}$, then by Claim 3.5, with probability at least $1 - n^{-12}$, we have $v \in X^{(t-1)}$, so v would *not* belong to $V^{(t)}$ and this means **no** $s \in S^{(t)}$ will satisfy $v \in B^{(t)}(s, r^{(t)})$, proving the claim for this case.

[5]In fact, for a given s, there is a unique t—if this s is ever chosen as a "starting point."

— *Case 2.* Otherwise, $|B^{(t-1)}(v, r^{(t)})| < n^{1-(t-1)/T}$. We have

$$|B^{(t)}(v, r^{(t)})| \leq |B^{(t-1)}(v, r^{(t)})| < n^{1-(t-1)/T}$$

as $B^{(t)}(v, r^{(t)}) \subseteq B^{(t-1)}(v, r^{(t)})$. Now let X be the number of s such that $v \in B^{(t)}(s, r^{(t)})$, so $X = \sum_{s \in S^{(t)}} \mathbf{1}_{\{s \in B^{(t)}(v, r^{(t)})\}}$. Over the random choice of $S^{(t)}$,

$$\mathbf{Pr}\left[s \in B^{(t)}(v, r^{(t)})\right] = \frac{|B^{(t)}(v, r^{(t)})|}{|V^{(t)}|} \leq \frac{1}{|V^{(t)}|} n^{1-(t-1)/T},$$

which gives

$$\mathbf{E}\left[X\right] = \sigma_t \cdot \mathbf{Pr}\left[s \in B^{(t)}(v, r^{(t)})\right] \leq 17 \log n.$$

To obtain a high probability bound for X, we will apply the tail bound in Lemma 6.2. Note that X is simply a hypergeometric random variable with the following parameters setting: total balls $N = |V^{(t)}|$, red balls $M = |B^{(t)}(v, r^{(t)})|$, and the number balls drawn is σ_t. Therefore, $\mathbf{Pr}\left[X \geq 34 \log n\right] \leq \exp\{-\frac{1}{4} \cdot 34 \log n\}$, so $X \leq 34 \log n$ with probability at least $1 - n^{-8}$.

Hence, regardless of what choices we made in rounds $1, 2, \ldots, t-2$, the conditional probability of seeing more than $34 \log n$ different s's is at most n^{-8}. Hence, we can remove the conditioning, and the claim follows. ∎

Lemma 6.9 *If for each vertex $u \in V$, there are at most $68 \log^2 n$ pairs (s, t) such that $s \in S^{(t)}$ and $u \in B^{(t)}(s, r^{(t)})$, then for an edge uv, the probability that u belongs to a different component than v is at most $68 \log^2 n / R$.*

Proof: We define a center $s \in S^{(t)}$ as "separating" u and v if $|B_s^{(t)} \cap \{u, v\}| = 1$. Clearly, if u, v lie in different components then there is some $t \in [T]$ and some center s that separates them. For a center $s \in S^{(t)}$, this can happen only if $\delta_s = R - \mathit{hop}(s, u)$, since $\mathit{hop}(s, v) \leq \mathit{hop}(s, u) - 1$. As there are R possible values of δ_s, this event occurs with probability at most $1/R$. And since there are only $68 \log^2 n$ different centers s that can possibly cut the edge, using a trivial union bound over them gives us an upper bound of $68 \log^2 n / R$ on the probability. ∎

To argue about (P3), notice that the premise to Lemma 6.9 holds with probability exceeding $1 - o(1) \geq 3/4$. Combining this with Lemma 6.6 proves property (P3), where the technical condition is the premise to Lemma 6.9.

Finally, we consider the work and depth of the algorithm. These are randomized bounds. Each computation of $B^{(t)}(v, r^{(t)})$ can be done using a BFS. Since $r^{(t)} \leq \rho$, the depth is bounded by $O(\rho \log n)$ per iteration, resulting in $O(\rho \log^2 n)$ after $T = O(\log n)$ iterations. As for work, by Lemma 6.6, each vertex is reached by at most $O(\log^2 n)$ starting points, yielding a total work of $O(m \log^2 n)$.

Low-Diameter Decomposition for Multiple Edge Classes

Extending the basic algorithm to support multiple edge classes is straightforward. The main idea is as follows. Suppose we are given a unweighted graph $G = (V, E)$, and the edge set E is composed of k edge classes $E_1 \uplus \cdots \uplus E_k$. So, if we run `splitGraph` on $G = (V, E)$ and ρ treating the different classes as one, then property (P3) indicates that each edge—regardless of which class it came from—is separated (i.e., it goes across components) with probability $p = \frac{136}{\rho} \log^3 n$. This allows us to prove the following corollary, which follows directly from Markov's inequality and the union bounds.

Corollary 6.10 *With probability at least $1/4$, for all $i \in [k]$, the number of edges in E_i that are between components is at most $|E_i| \frac{272k \log^3 n}{\rho}$.*

The corollary suggests a simple way to use `splitGraph` to provide guarantees required by Theorem 6.3: as summarized in Algorithm 6.4.2, we run `splitGraph` on the input graph treating all edge classes as one and repeat it if any of the edge classes had too many edges cut (i.e., more than $|E_i| \frac{272k \log^3 n}{\rho}$). As the corollary indicates, the number of trials is a geometric random variable with with $p = 1/4$, so in expectation, it will finish after 4 trials. Furthermore, although it could go on forever in the worst case, the probability does fall exponentially fast.

Algorithm 6.4.2 `Partition` $(G = (V, E = E_1 \uplus \cdots \uplus E_k), \rho)$ — Partition an input graph G into components of radius at most ρ.

```
1. Let $\mathcal{C}$ = splitGraph($(V, \uplus E_i), \rho$).
2. If there is some $i$ such that $E_i$ has more than $|E_i| \frac{272 \cdot k \log^3 n}{\rho}$ edges between components, start over. (Recall that $k$ was the number of edge classes.)
Return $\mathcal{C}$.
```

Finally, we note that properties (P1) and (P2) directly give Theorem 6.3(1)–(2)—and the validation step in `Partition` ensures Theorem 6.3(3), setting $c_1 = 272$. The work and depth bounds for `Partition` follow from the bounds derived for `splitGraph` and Corollary 6.10. This concludes the proof of Theorem 6.3.

6.5 Parallel Low-Stretch Spanning Trees and Subgraphs

This section presents parallel algorithms for low-stretch spanning trees and for low-stretch spanning subgraphs. To obtain the low-stretch spanning tree algorithm, we apply the construction of Alon et al. [AKPW95] (henceforth, the AKPW construction), together with the parallel graph partition algorithm from the previous section. The resulting procedure, however, is not ideal for two reasons: the depth of the algorithm depends on the "spread" Δ—the ratio between the heaviest edge and the lightest edge—and even for polynomial spread, both the depth and the average stretch are super-logarithmic (both of them have a $2^{O(\sqrt{\log n \cdot \log\log n})}$ term). Fortunately, for our application, we observe that we do not need spanning trees but merely low-stretch sparse graphs. In Section 6.5.2, we describe modifications to this construction to obtain a parallel algorithm which computes sparse subgraphs that give us only polylogarithmic average stretch and that can be computed in polylogarithmic depth and $\widetilde{O}(m)$ work. We believe that this construction may be of independent interest.

6.5.1 Low-Stretch Spanning Trees

Using the AKPW construction, along with the `Partition` procedure from Section 6.4, we will prove the following theorem:

Theorem 6.11 (Low-Stretch Spanning Tree) *There is an algorithm* `AKPW`(G) *which given as input a graph* $G = (V, E, w)$*, produces a spanning tree in* $O(\log^{O(1)} n \cdot 2^{O(\sqrt{\log n \cdot \log\log n})} \log \Delta)$ *expected depth and* $\widetilde{O}(m)$ *expected work such that the total stretch of all edges is bounded by* $m \cdot 2^{O(\sqrt{\log n \cdot \log\log n})}$.

Presented in Algorithm 6.5.1 is a restatement of the AKPW algorithm, except that here we will use our parallel low-diameter decomposition for the partition step. In words, iteration j of Algorithm 6.5.1 looks at a graph $(V^{(j)}, E^{(j)})$ which is a minor of the original graph (because components were contracted in previous iterations, and because it only considers the edges in the first j weight classes). It uses `Partition`$((V, \uplus_{j \le k} E_j), z/4)$ to decompose this graph into components such that the hop radius is at most $z/4$ and each weight class has only $1/y$ fraction of its edges crossing between components. (Parameters y, z are defined in the algorithm and are slightly different from the original settings in the AKPW algorithm.) It then shrinks each of the components into a single node (while adding a BFS tree on that component to T), and iterates on this graph. Adding these BFS trees maintains the invariant

Algorithm 6.5.1 `AKPW` $(G = (V, E, w))$ — a low-stretch spanning tree construction.

```
i. Normalize the edges so that $\min\{w(e) : e \in E\} = 1$.
ii. Let $y = 2^{\sqrt{6 \log n \cdot \log\log n}}$, $\tau = \lceil 3 \log(n)/\log y \rceil$, $z = 4c_1 y \tau \log^3 n$. Initialize $T = \emptyset$.
iii. Divide $E$ into $E_1, E_2, \ldots$, where $E_i = \{e \in E \mid w(e) \in [z^{i-1}, z^i)\}$.
     Let $E^{(1)} = E$ and $E_i^{(1)} = E_i$ for all $i$.
iv. For $j = 1, 2, \ldots$, until the graph is exhausted,
    1. $(C_1, C_2, \ldots, C_p) =$ Partition$((V^{(j)}, \uplus_{i \le j} E_i^{(j)}), z/4)$
    2. Add a BFS tree of each component to $T$.
    3. Define graph $(V^{(j+1)}, E^{(j+1)})$ by contracting all edges within the components and removing all self-loops (but maintaining parallel edges). Create $E_i^{(j+1)}$ from $E_i^{(j)}$ taking into account the contractions.
v. Output the tree $T$.
```

that the set of original nodes which have been contracted into a (super-)node in the current graph are connected in T; hence, when the algorithm stops, we have a spanning tree of the original graph—hopefully of low total stretch.

We begin the analysis of the total stretch and running time by proving two useful facts:

Fact 6.12 *The number of edges $|E_i^{(j)}|$ is at most $|E_i|/y^{j-i}$.*

Proof: If we could ensure that the number of weight classes in play at any time is at most τ, the number of edges in each class would fall by at least a factor of $\frac{c_1 \tau \log^3 n}{z/4} = 1/y$ by Theorem 6.3(3) and the definition of z, and this would prove the fact. Now, for the first τ iterations, the number of weight classes is at most τ just because we consider only the first j weight classes in iteration j. Now in iteration $\tau + 1$, the number of surviving edges of E_1 would fall to $|E_1|/y^\tau \le |E_1|/n^3 < 1$, and hence there would only be τ weight classes left. It is easy to see that this invariant can be maintained over the course of the algorithm. ■

Fact 6.13 *In iteration j, the radius of a component according to edge weights (in the expanded-out graph) is at most z^{j+1}.*

Proof: The proof is by induction on j. First, note that by Theorem 6.3(2), each of the clusters computed in any iteration j has edge-count radius at most $z/4$. Now the base case $j = 1$ follows by noting that each edge in E_1 has weight less than z, giving a radius of at most $z^2/4 < z^{j+1}$. Now assume inductively that the radius in iteration $j - 1$ is at most z^j. Now any path with $z/4$ edges from the center to some node in the contracted graph will pass through at most $z/4$ edges of weight at most z^j, and at most $z/4 + 1$ supernodes, each of which adds a distance of $2z^j$; hence, the new radius is at most $z^{j+1}/4+(z/4+1)2z^j \leq z^{j+1}$ as long as $z \geq 8$. ■

Applying these facts, we bound the total stretch of an edge class.

Lemma 6.14 *For any $i \geq 1$, $\text{str}_T(E_i) \leq 4y^2|E_i|(4c_1\tau\log^3 n)^{\tau+1}$.*

Proof: Let e be an edge in E_i contracted during iteration j. Since $e \in E_i$, we know $w(e) > z^{i-1}$. By Fact 6.13, the path connecting the two endpoints of e in F has distance at most $2z^{j+1}$. Thus, $\text{str}_T(e) \leq 2z^{j+1}/z^{i-1} = 2z^{j-i+2}$. Fact 6.12 indicates that the number of such edges is at most $|E_i^{(j)}| \leq |E_i|/y^{j-i}$. We conclude that

$$\begin{aligned}\text{str}_T(E_i) &\leq \sum_{j=i}^{i+\tau-1} 2z^{j-i+2}|E_i|/y^{j-i}\\ &\leq 4y^2|E_i|(4c_1\tau\log^3 n)^{\tau+1}\end{aligned}$$

■

Proof of Theorem 6.11: Summing across the edge classes gives the promised bound on stretch. Now there are $\lceil\log_z \Delta\rceil$ weight classes E_i's in all, and since each time the number of edges in a (non-empty) class drops by a factor of y, the algorithm has at most $O(\log\Delta+\tau)$ iterations. By Theorem 6.3 and standard techniques, each iteration does $O(m\log^2 n)$ work and has $O(z\log^2 n) = O(\log^{O(1)} n \cdot 2^{O(\sqrt{\log n\cdot\log\log n})})$ depth in expectation. ■

6.5.2 Low-Stretch Spanning Subgraphs

We now show how to alter the parallel low-stretch spanning tree construction from the preceding section to give a low-stretch spanning *subgraph* construction that has no dependence on the "spread," and moreover has only polylogarithmic stretch. This comes at the cost of obtaining a sparse subgraph with $n-1+O(m/\operatorname{polylog} n)$ edges instead of a tree, but suffices for our solver application. The two main ideas behind these improvements are the following: Firstly, the number of surviving edges in each weight class decreases by a logarithmic factor in each iteration; hence, we could throw in all surviving edges after they have been whittled down in a constant number of iterations—this removes the factor of $2^{O(\sqrt{\log n\cdot\log\log n})}$from both the average stretch and the depth. Secondly, if Δ is large, we will identify certain weight-classes with $O(m/\operatorname{polylog} n)$ edges, which by setting them aside, will allow us to break up the chain of dependencies and obtain $O(\operatorname{polylog} n)$ depth; these edges will be thrown back into the final solution, adding $O(m/\operatorname{polylog} n)$ extra edges (which we can tolerate) without increasing the average stretch.

The First Improvement

Let us first show how to achieve polylogarithmic stretch with an ultra-sparse subgraph. Given parameters $\lambda \in \mathbb{Z}_{>0}$ and $\beta \geq c_2 \log^3 n$ (where $c_2 = 2 \cdot (4c_1(\lambda+1))^{\frac{1}{2}(\lambda-1)}$), we obtain the new algorithm $\texttt{SparseAKPW}(G, \lambda, \beta)$ by modifying Algorithm 6.5.1 as follows:

(1) use the altered parameters $y = \frac{1}{c_2}\beta/\log^3 n$ and $z = 4c_1 y(\lambda+1)\log^3 n$;

(2) in each iteration j, call $\texttt{Partition}$ with at most $\lambda+1$ edge classes—keep the λ classes $E_j^{(j)}, E_{j-1}^{(j)}, \ldots, E_{j-\lambda+1}^{(j)}$, but then define a "generic bucket" $E_0^{(j)} := \cup_{j' \leq j-\lambda} E_{j'}^{(j)}$ as the last part of the partition; and

(3) finally, output not just the tree T but the subgraph $\widehat{G} = T \cup (\cup_{i \geq 1} E_i^{(i+\lambda)})$.

Lemma 6.15 *Given a graph G, parameters $\lambda \in \mathbb{Z}_{>0}$ and $\beta \geq c_2 \log^3 n$ (where $c_2 = 2 \cdot (4c_1(\lambda+1))^{\frac{1}{2}(\lambda-1)}$) the algorithm SparseAKPW(G, λ, β) outputs a subgraph of G with at most $n - 1 + m(c_2(\log^3 n/\beta))^\lambda$ edges and total stretch at most $m\beta^2 \log^{3\lambda+3} n$. Moreover, the expected work is $\widetilde{O}(m)$ and expected depth is*
$O((c_1\beta/c_2)\lambda \log^2 n(\log \Delta + \log n))$.

Proof: The proof parallels that for Theorem 6.11. Fact 6.13 remains unchanged. The claim from Fact 6.12 now remains true only for $j \in \{i, \ldots, i+\lambda-1\}$; after that the edges in $E_i^{(j)}$ become part of $E_0^{(j)}$, and we only give a cumulative guarantee on the generic bucket. But this does hurt us: if $e \in E_i$ is contracted in iteration $j \leq i+\lambda-1$ (i.e., it lies within a component formed in iteration j), then $\mathsf{str}_{\widehat{G}}(e) \leq 2z^{j-i+2}$. And the edges of E_i that survive till iteration $j \geq i+\lambda$ have stretch 1 because they are eventually all added to $\widehat{G}$; hence we do not have to worry that they belong to the class $E_0^{(j)}$ for those iterations. Thus, $\mathsf{str}_{\widehat{G}}(E_i) \leq \sum_{j=i}^{i+\lambda-1} 2z^{j-i+2} \cdot |E_i|/y^{j-i} \leq 4y^2(\frac{z}{y})^{\lambda-1}|E_i|$.

Summing across the edge classes gives $\mathsf{str}_{\widehat{G}}(E) \leq 4y^2(\frac{z}{y})^{\lambda-1}m$, which simplifies to $O(m\beta^2 \log^{3\lambda+3} n)$. Next, the number of edges in the output follows directly from the fact T can have at most $n-1$ edges, and the number of extra edges from each class is only a $1/y^\lambda$ fraction (i.e., $|E_i^{(i+\lambda)}| \leq |E_i|/y^\lambda$ from Fact 6.12). Finally, the work remains the same; for each of the $(\log \Delta + \tau)$ distance scales the depth is still $O(z \log^2 n)$, but the new value of z causes this to become $O((c_1\beta/c_2)\lambda \log^2 n)$. ■

The Second Improvement

The depth of the `SparseAKPW` algorithm still depends on $\log \Delta$, and the reason is straightforward: the graph $G^{(j)}$ used in iteration j is built by taking $G^{(1)}$ and contracting edges in each iteration—hence, it depends on all previous iterations. However, the crucial observation is that if we had τ consecutive weight classes E_i's which are empty, we could break this chain of dependencies at this point. However, there may be no empty weight classes; but having weight classes with relatively few edges is enough, as we show next.

Fact 6.16 *Given a graph $G = (V, E)$ and a subset of edges $F \subseteq E$, let $G' = G \setminus F$ be a potentially disconnected graph. If $\widehat{G}'$ is a subgraph of G' with total stretch $\mathrm{str}_{\widehat{G}'}(E(G')) \leq D$, then the total stretch of E on $\widehat{G} := \widehat{G}' \cup F$ is at most $|F| + D$.*

Consider a graph $G = (V, E, w)$ with edge weights $w(e) \geq 1$, and let $E_i(G) := \{e \in E(G) \mid w(e) \in [z^{i-1}, z^i)\}$ be the weight classes. Then, G is called (γ, τ)-*well-spaced* if there is a set of *special* weight classes $\{E_i(G)\}_{i \in I}$ such that for each $i \in I$, (a) there are at most γ weight classes before the following special weight class $\min\{i' \in I \cup \{\infty\} \mid i' > i\}$, and (b) the τ weight classes $E_{i-1}(G), E_{i-2}(G), \ldots, E_{i-\tau}(G)$ preceding i are all empty.

Lemma 6.17 *Given any graph $G = (V, E)$, $\tau \in \mathbb{Z}_+$, and $\theta \leq 1$, there exists a graph $G' = (V, E')$ which is $(4\tau/\theta, \tau)$-well-spaced, and $|E' \setminus E| \leq \theta \cdot |E|$. Moreover, G' can be constructed in $O(m)$ work and $O(\log n)$ depth.*

Proof: Let $\delta = \frac{\log \Delta}{\log z}$; note that the edge classes for G are $E_1, \ldots, E_\delta$, some of which may be empty. Denote by E_J the union $\cup_{i \in J} E_i$. We construct G' as follows: Divide these edge classes into disjoint groups $J_1, J_2, \ldots \subseteq [\delta]$, where each group consists of $\lceil \tau/\theta \rceil$ consecutive classes. Within a group J_i, by an averaging argument, there must be a range $L_i \subseteq J_i$ of τ *consecutive* edge classes that contains at most a θ fraction of all the edges in this group, i.e., $|E_{L_i}| \leq \theta \cdot |E_{J_i}|$ and $|L_i| \geq \tau$. We form G' by removing these the edges in all these groups L_i's from G, i.e., $G' = (V, E \setminus (\cup_i E_{L_i}))$. This removes only a θ fraction of all the edges of the graph.

We claim G' is $(4\tau/\theta, \tau)$-well-spaced. Indeed, if we remove the group L_i, then we designate the smallest $j \in [\delta]$ such that $j > \max\{j' \in L_i\}$ as a special bucket (if such a j exists). Since we removed the edges in E_{L_i}, the second condition for being well-spaced follows. Moreover, the number of buckets between a special bucket and the following one is at most $2\lceil \tau/\theta \rceil - (\tau - 1) \leq 4\tau/\theta$. Finally, these computations can be done in $O(m)$ work and $O(\log n)$ depth using standard techniques [JáJ92, Lei92]. ■

Lemma 6.18 *Let $\tau = 3\log n/\log y$. Given a graph G which is (γ, τ)-well-spaced, SparseAKPW can be computed on G with $\widetilde{O}(m)$ work and $O(\frac{c_1}{c_2}\gamma\lambda\beta\log^2 n)$ depth.*

Proof: Since G is (γ, τ)-well-spaced, each special bucket $i \in I$ must be preceded by τ empty buckets. Hence, in iteration i of `SparseAKPW`, any surviving edges belong to buckets $E_{i-\tau}$ or smaller. However, these edges have been reduced by a factor of y in each iteration and since $\tau > \log_y n^2$, all the edges have been contracted in previous iterations—i.e., $E_\ell^{(i)}$ for $\ell < i$ is empty.

Consider any special bucket i: we claim that we can construct the vertex set $V^{(i)}$ that `SparseAKPW` sees at the beginning of iteration i, without having to run the previous iterations. Indeed, we can just take the MST on the entire graph $G = G^{(1)}$, retain only the edges from buckets $E_{i-\tau}$ and lower, and contract the connected components of this forest to get $V^{(i)}$. And once we know this vertex set $V^{(i)}$, we can drop out the edges from E_i and higher buckets which have been contracted (these are now self-loops), and execute iterations $i, i+1, \ldots$ of `SparseAKPW` without waiting for the preceding iterations to finish. Moreover, given the MST, all this can be done in $O(m)$ work and $O(\log n)$ depth.

Finally, for each special bucket i in parallel, we start running `SparseAKPW` at iteration i. Since there are at most γ iterations until the next special bucket, the total depth is only $O(\gamma z \log^2 n) = O(\frac{c_1}{c_2}\gamma\lambda\beta\log^2 n)$. ■

Theorem 6.19 (Low-Stretch Subgraphs) *Given a weighted graph G, $\lambda \in \mathbb{Z}_{>0}$, and $\beta \geq c_2\log^3 n$ (where $c_2 = 2\cdot(4c_1(\lambda+1))^{\frac{1}{2}(\lambda-1)}$), there is an algorithm LSSubgraph(G, β, λ) that finds a subgraph $\widehat{G}$ such that*

1. *$|E(\widehat{G})| \leq n - 1 + m\left(c_{LS}\frac{\log^3 n}{\beta}\right)^\lambda$*
2. *The total stretch (of all $E(G)$ edges) in the subgraph $\widehat{G}$ is at most by $m\beta^2\log^{3\lambda+3} n$,*

where c_{LS} $(= c_2 + 1)$ is a constant. Moreover, the procedure runs in $O(\lambda\beta^{\lambda+1}\log^{3-3\lambda} n)$ depth and $\widetilde{O}(m)$ work. If $\lambda = O(1)$ and $\beta = \mathrm{polylog}(n)$, the depth term simplifies to $O(\log^{O(1)} n)$.

Proof: Given a graph G, we set $\tau = 3\log n/\log y$ and $\theta = (\log^3 n/\beta)^\lambda$, and apply Lemma 6.17 to delete at most θm edges, and get a $(4\tau/\theta, \tau)$-well-spaced graph G'. Let $m' = |E'|$. On this graph, we run `SparseAKPW` to obtain a graph $\widehat{G}'$ with $n - 1 + m'(c_2(\log^3 n/\beta))^\lambda$ edges and total stretch at most $m'\beta^2\log^{3\lambda+3} n$; moreover, Lemma 6.18 shows this can be computed with $\widetilde{O}(m)$ work and the depth is

$$O(\frac{c_1}{c_2}(4\tau/\theta)\lambda\beta\log^2 n) = O(\lambda\beta^{\lambda+1}\log^{3-3\lambda} n).$$

Finally, we output the graph $\widehat{G} = \widehat{G}' \cup (E(G) \setminus E(G'))$; this gives the desired bounds on stretch and the number of edges as implied by Fact 6.16 and Lemma 6.15. ■

6.6 Parallel SDD Solver

In this section, we derive a parallel solver for symmetric diagonally dominant (SDD) linear systems, using the ingredients developed in the previous sections. The solver follows closely the line of work of [ST03, ST06, KM07, KMP10]. Specifically, we will derive a proof for the main theorem (Theorem 6.1), the statement of which is reproduced below.

> **Theorem 6.1.** For any fixed $\theta > 0$ and any $\varepsilon > 0$, there is an algorithm `SDDSolve` that on input an SDD matrix A and a vector b computes a vector $\tilde{x}$ such that $\|\tilde{x} - A^+b\|_A \leq \varepsilon \cdot \|A^+b\|_A$ in $O(m\log^{O(1)} n \log\frac{1}{\varepsilon})$ work and $O(m^{1/3+\theta}\log\frac{1}{\varepsilon})$ depth.

In proving this theorem, we will focus on Laplacian linear systems. As noted earlier, linear systems on SDD matrices are reducible to systems on graph Laplacians in $O(\log(m+n))$ depth and $O(m+n)$ work [Gre96]. Furthermore, because of the one-to-one correspondence between graphs and their Laplacians, we will use the two terms interchangeably.

The core of the near-linear time Laplacian solvers in [ST03, ST06, KMP10] is a "preconditioning" chain of progressively smaller graphs $\langle A_1 = A, A_2, \ldots, A_d\rangle$, along with a well-understood recursive algorithm, known as recursive preconditioned Chebyshev method—`rPCh`, that traverses the levels of the chain and for each visit at level $i < d$, performs $O(1)$ matrix-vector multiplications with A_i and other simple vector-vector operations. Each time the algorithm reaches level d, it solves a linear system on A_d using a direct method. Except for solving the bottom-level systems, all these operations can be accomplished in linear work and $O(\log(m+n))$ depth. The recursion itself is based on a simple scheme; for each visit at level i the algorithm makes at most κ'_i recursive calls to level $i+1$, where $\kappa'_i \geq 2$ is a fixed system-independent integer. Therefore, assuming we have computed a chain of preconditioners, the total required depth is (up to a log) equal to the total number of times the algorithm reaches the last (and smallest) level A_d.

6.6.1 Parallel Construction of Solver Chain

The construction of the preconditioning chain in [KMP10] relies on a subroutine that on input a graph A_i, constructs a slightly sparser graph B_i which is spectrally related to A_i. This "incremental sparsification" routine is in turn based on the computation of a low-stretch tree for A_i. The parallelization of the low-stretch tree is actually the main obstacle in parallelizing the whole solver presented in [KMP10]. Crucial to effectively applying our result in Section 6.5 is a simple observation that the sparsification routine of [KMP10] only requires a low-stretch spanning subgraph rather than a tree.Then, with the exception of some parameters in its construction, the preconditioning chain remains essentially the same.

The following lemma is immediate from Section 6 of [KMP10].

Lemma 6.20 *Given a graph G and a subgraph $\widehat{G}$ of G such that the total stretch of all edges in G with respect to $\widehat{G}$ is $m \cdot S$, a parameter on condition number κ, and a success probability $1 - 1/\xi$, there is an algorithm that constructs a graph H such that*

1. $G \preceq H \preceq \kappa \cdot G$, *and*
2. $|E(H)| = |E(\widehat{G})| + (c_{IS} \cdot S \log n \log \xi)/\kappa$

in $O(\log^2 n)$ depth and $O(m \log^2 n)$ work, where c_{IS} is an absolute constant.

Although Lemma 6.20 was originally stated with $\widehat{G}$ being a spanning tree, the proof in fact works without changes for an arbitrary subgraph. For our purposes, ξ has to be at most $O(\log n)$ and that introduces an additional $O(\log \log n)$ term. For simplicity, in the rest of the section, we will consider this as an extra $\log n$ factor.

Lemma 6.21 *Given a weighted graph G, parameters λ and η such that $\eta \geq \lambda \geq 16$, we can construct in $O(\log^{2\eta\lambda} n)$ depth and $\widetilde{O}(m)$ work another graph H such that*

1. $G \preceq H \preceq \frac{1}{10} \cdot \log^{\eta\lambda} n \cdot G$
2. $|E(H)| \leq n - 1 + m \cdot c_{PC}/\log^{\eta\lambda - 2\eta - 4\lambda}(n)$,

where c_{PC} is an absolute constant.

Proof: Let $\widehat{G} = \texttt{LSSubgraph}(G, \lambda, \log^\eta n)$. Then, Theorem 6.19 shows that $|E(\widehat{G})|$ is at most

$$n - 1 + m \left(\frac{c_{LS} \cdot \log^3 n}{\beta} \right)^\lambda = n - 1 + m \left(\frac{c_{LS}}{\log^{\eta - 3} n} \right)^\lambda$$

Furthermore, the total stretch of all edges in G with respect to $\widehat{G}$ is at most

$$S = m\beta^2 \log^{\lambda+3} n \leq m \log^{2\eta+3\lambda+3} n.$$

Applying Lemma 6.20 with $\kappa = \frac{1}{10}\log^{\eta\lambda} n$ gives H such that $G \preceq H \preceq \frac{1}{10}\log^{\eta\lambda} n \cdot G$ and $|E(H)|$ is at most

$$\begin{aligned}
& n - 1 + m \cdot \left(\frac{c_{LS}^{\lambda}}{\log^{\lambda(\eta-3)} n} + \frac{10 \cdot c_{IS} \log^{2\eta+3\lambda+5} n}{\log^{\eta\lambda} n} \right) \\
\leq \quad & n - 1 + m \cdot \frac{c_{PC}}{\log^{\eta\lambda-2\lambda-3k-5} n} \\
\leq \quad & n - 1 + m \cdot \frac{c_{PC}}{\log^{\eta\lambda-2\eta-4\lambda} n}.
\end{aligned}$$

■

We now give a more precise definition of the preconditioning chain we use for the parallel solver by giving the pseudocode for constructing it.

Definition 6.22 (Preconditioning Chain) *Consider a chain of graphs*

$$\mathcal{C} = \langle A_1 = A, B_1, A_2, \ldots, A_d \rangle,$$

and denote by n_i and m_i the number of nodes and edges of A_i respectively. We say that $\mathcal{C}$ is preconditioning chain *for A if*

1. *$B_i = \texttt{IncrementalSparsify}(A_i)$.*
2. *$A_{i+1} = \texttt{GreedyElimination}(B_i)$.*
3. *$A_i \preceq B_i \preceq 1/10 \cdot \kappa_i A_i$, for some explicitly known integer κ_i.* [6]

As noted above, the `rPCh` algorithm relies on finding the solution of linear systems on A_d, the bottom-level systems. To parallelize these solves, we make use of the following fact which can be found in Sections 3.4. and 4.2 of [GVL96].

Fact 6.23 *A factorization $LL^\top$ of the pseudo-inverse of an n-by-n Laplacian A, where L is a lower triangular matrix, can be computed in $O(n)$ time and $O(n^3)$ work, and any solves thereafter can be done in $O(\log n)$ time and $O(n^2)$ work.*

[6]The constant of $1/10$ in the condition number is introduced only to simplify subsequent notation.

Note that although A is not positive definite, its null space is the space spanned by the all 1s vector when the underlying graph is connected. Therefore, we can in turn drop the first row and column to obtain a semi-definite matrix on which LU factorization is numerically stable.

The routine `GreedyElimination` is a partial Cholesky factorization (for details see [ST06] or [KMP10]) on vertices of degree at most 2. From a graph-theoretic point of view, the routine `GreedyElimination` can be viewed as simply recursively removing nodes of degree one and splicing out nodes of degree two. The sequential version of `GreedyElimination` returns a graph with no degree 1 or 2 nodes. The parallel version that we present below leaves some degree-2 nodes in the graph, but their number will be small enough to not affect the complexity.

Lemma 6.24 *If G has n vertices and $n-1+m$ edges, then the procedure `GreedyElimination`(G) returns a graph with at most $2m-2$ nodes in $O(n+m)$ work and $O(\log n)$ depth* ***whp****.*

Proof: The sequential version of `GreedyElimination`(G) is equivalent to repeatedly removing degree 1 vertices and splicing out 2 vertices until no more exist while maintaining self-loops and multiple edges (see, e.g., [ST03, ST06] and [Kou07, Section 2.3.4]). Thus, the problem is a slight generalization of parallel tree contraction [MR89]. In the parallel version, we show that while the graph has more than $2m-2$ nodes, we can efficiently find and eliminate a "large" independent set of degree two nodes, in addition to all degree one vertices.

We alternate between two steps, which are equivalent to `Rake` and `Compress` in [MR89], until the vertex count is at most $2m-2$:
Mark an independent set of degree 2 vertices, then

1. Contract all degree 1 vertices, and
2. Compress and/or contract out the marked vertices.

To find the independent set, we use a randomized marking algorithm on the degree two vertices (this is used in place of maximal independent set for work efficiency): Each degree two node flips a coin with probability $\frac{1}{3}$ of turning up heads; we mark a node if it is a heads and its neighbors either did not flip a coin or flipped a tail.

We show that the two steps above will remove a constant fraction of "extra" vertices. Let G is a multigraph with n vertices and $m+n-1$ edges. First, observe that if all vertices have degree at least three then $n \leq 2(m-1)$ and we would be finished. So, let T be any fixed spanning tree of G; let a_1 (resp. a_2) be the number of vertices in T of degree one (resp. two) and a_3 the number those of degree three or more. Similarly, let b_1, b_2, and b_3 be the number vertices in G of degree 1, 2, and at least 3, respectively, where the degree is the vertex's degree in G.

It is easy to check that in expectation, these two steps remove $b_1 + \frac{4}{27}b_2 \geq b_1 + \frac{1}{7}b_2$ vertices. In the following, we will show that $b_1 + \frac{1}{7}b_2 \geq \frac{1}{7}\Delta n$, where $\Delta n = n - (2m-2) = n - 2m + 2$ denotes the number of "extra" vertices in the graph. Consider non-tree edges and how they are attached to the tree T. Let m_1, m_2, and m_3 be the number of attachment of the following types, respectively:

(1) an attachment to x, a degree 1 vertex in T, where x has at least one other attachment.
(2) an attachment to x, a degree 1 vertex in T, where x has no other attachment.
(3) an attachment to a degree 2 vertex in T.

As each edge is incident on two endpoints, we have $m_1 + m_2 + m_3 \leq 2m$. Also, we can lower bound b_1 and b_2 in terms of m_i's and a_i's: we have $b_1 \geq a_1 - m_1/2 - m_2$ and $b_2 \geq m_2 + a_2 - m_3$. This gives

$$\begin{aligned} b_1 + \tfrac{1}{7}b_2 &\geq \tfrac{2}{7}(a_1 - m_1/2 - m_2) + \tfrac{1}{7}(m_2 + a_2 - m_3) \\ &= \tfrac{2}{7}a_1 + \tfrac{1}{7}a_2 - \tfrac{1}{7}(m_1 + m_2 + m_3) \\ &\geq \tfrac{2}{7}a_1 + \tfrac{1}{7}a_2 - \tfrac{2}{7}m. \end{aligned}$$

Consequently, $b_1 + \frac{1}{7}b_2 \geq \frac{1}{7}(2a_1 + a_2 - 2m) \geq \frac{1}{7} \cdot \Delta n$, where to show the last step, it suffices to show that $n + 2 \leq 2a_1 + a_2$ for a tree T of n nodes. WLOG, we may assume that all nodes of T have degree either one or three, in which case $2a_1 = n + 2$. Finally, by Chernoff bounds, the algorithm will finish with high probability in $O(\log n)$ rounds. ∎

6.6.2 Parallel Performance of Solver Chain

Spielman and Teng [ST06, Section 5] gave a (sequential) time bound for solving a linear SDD system given a preconditioner chain. The following lemma extends their Theorem 5.5 to give parallel runtime bounds (work and depth), as a function of κ_i's and m_i's. We note that in the bounds below, the m_d^2 term arises from the dense inverse used to solve the linear system in the bottom level.

Lemma 6.25 *There is an algorithm that given a preconditioner chain $\mathcal{C} = \langle A_1 = A, A_2, \ldots, A_d \rangle$ for a matrix A, a vector b, and an error tolerance ε, computes a vector $\tilde{x}$ such that*

$$\|\tilde{x} - A^+ b\|_A \;\le\; \varepsilon \cdot \|A^+ b\|_A,$$

with depth bounded by

$$\left(\sum_{1\le i\le d}\prod_{1\le j<i}\sqrt{\kappa_j}\right)\log n \log\left(\tfrac{1}{\varepsilon}\right) \;\le\; O\left(\left(\prod_{1\le j<d}\sqrt{\kappa_j}\right)\log n\log\left(\tfrac{1}{\varepsilon}\right)\right)$$

and work bounded by

$$\left(\sum_{1\le i\le d-1} m_i \cdot \prod_{j\le i}\sqrt{\kappa_j} + m_d^2 \prod_{1\le j<d}\sqrt{\kappa_j}\right)\log\left(\tfrac{1}{\varepsilon}\right).$$

To reason about Lemma 6.25, we will rely on the following lemma about preconditioned Chebyshev iteration and the recursive solves that happen at each level of the chain. This lemma is a restatement of Spielman and Teng's Lemma 5.3 (slightly modified so that the $\sqrt{\kappa_i}$ does not involve a constant, which shows up instead as constant in the preconditioner chain's definition).

Lemma 6.26 *Given a preconditioner chain of length d, it is possible to construct linear operators $\texttt{solve}_{A_i}$ for all $i \le d$ such that*

$$(1 - e^{-2})A_i^+ \preceq solve_{A_i} \preceq (1 + e^2)$$

and $\texttt{solve}_{A_i}$ is a polynomial of degree $\sqrt{\kappa_i}$ involving $\texttt{solve}_{A_{i+1}}$ and 4 matrices with m_i non-zero entries (from `GreedyElimination`).

Armed with this, we state and prove the following lemma:

Lemma 6.27 *For $\ell \ge 1$, given any vector b, the vector $\texttt{solve}_{A_\ell} \cdot b$ can be computed in depth*

$$\log n \sum_{\ell\le i\le d}\prod_{\ell\le j<i}\sqrt{\kappa_j}$$

and work

$$\sum_{\ell\le i\le d-1} m_i \cdot \prod_{\ell\le j\le i}\sqrt{\kappa_j} + m_d^2 \prod_{\ell\le j<d}\sqrt{\kappa_j}$$

Proof: The proof is by induction in decreasing order on ℓ. When $d = \ell$, all we are doing is a matrix multiplication with a dense inverse. This takes $O(\log n)$ depth and $O(m_d^2)$ work.

Suppose the result is true for $\ell + 1$. Then since $\texttt{solve}_{A_\ell}$ can be expressed as a polynomial of degree $\sqrt{\kappa_\ell}$ involving an operator that is $\texttt{solve}_{A_{\ell+1}}$ multiplied by at most 4 matrices with $O(m_\ell)$ non-zero entries. We have that the total depth is

$$\begin{aligned}
&\log n\sqrt{\kappa_\ell} + \sqrt{\kappa_\ell} \cdot \left(\log n \sum_{\ell+1 \le i \le d} \prod_{\ell+1 \le j < i} \sqrt{\kappa_j}\right) \\
&= \log n \sum_{\ell \le i \le d} \prod_{\ell \le j < i} \sqrt{\kappa_j}
\end{aligned}$$

and the total work is bounded by

$$\begin{aligned}
&\sqrt{\kappa_\ell} m_\ell + \sqrt{\kappa_\ell} \cdot \left(\sum_{\ell+1 \le i \le d-1} m_i \cdot \prod_{\ell+1 \le j \le i} \sqrt{\kappa_j} + m_d^2 \prod_{\ell+1 \le j < d} \sqrt{\kappa_j} \right) \\
&= \sum_{\ell \le i \le d-1} m_i \cdot \prod_{\ell \le j \le i} \sqrt{\kappa_j} + m_d^2 \prod_{\ell \le j < d} \sqrt{\kappa_j}.
\end{aligned}$$

■

Proof of Lemma 6.25: The ε-accuracy bound follows from applying preconditioned Chebyshev to $\texttt{solve}_{A_1}$ similarly to Spielman and Teng's Theorem 5.5 [ST06], and the running time bounds follow from Lemma 6.27 when $\ell = 1$. ■

6.6.3 Optimizing the Chain for Depth

Lemma 6.25 shows that the algorithm's performance is determined by the settings of κ_i's and m_i's; however, as we will be using Lemma 6.21, the number of edges m_i is essentially dictated by our choice of κ_i. We now show that if we terminate chain earlier, i.e. adjusting the dimension A_d to roughly $O(m^{1/3} \log \varepsilon^{-1})$, we can obtain good parallel performance. As a first attempt, we will set κ_i's uniformly:

Lemma 6.28 *For any fixed $\theta > 0$, if we construct a preconditioner chain using Lemma 6.21 setting λ to some proper constant greater than 21, $\eta = \lambda$ and extending the sequence until $m_d \le m^{1/3-\delta}$ for some δ depending on λ, we get a solver algorithm that runs in $O(m^{1/3+\theta} \log(1/\varepsilon))$ depth and $\widetilde{O}(m \log 1/\varepsilon)$ work as $\lambda \to \infty$, where ε is the accuracy precision of the solution, as defined in the statement of Theorem 6.1.*

Proof: By Lemma 6.20, we have that m_{i+1}—the number of edges in level $i+1$—is bounded by

$$O(m_i \cdot \frac{c_{PC}}{\log^{\eta\lambda-2\eta-4\lambda}}) = O(m_i \cdot \frac{c_{PC}}{\log^{\lambda(\lambda-6)}}),$$

which can be repeatedly apply to give

$$m_i \leq m \cdot \left(\frac{c_{PC}}{\log^{\lambda(\lambda-6)} n}\right)^{i-1}$$

Therefore, when $\lambda > 12$, we have that for each $i < d$,

$$\begin{aligned}
m_i \cdot \prod_{j \leq i} \sqrt{\kappa(n_j)} &\leq m \cdot \left(\frac{c_{PC}}{\log^{\lambda(\lambda-6)} n}\right)^{i-1} \cdot \left(\sqrt{\log^{\lambda^2} n}\right)^{i} \\
&= \tilde{O}(m) \cdot \left(\frac{c_{PC}}{\log^{\lambda(\lambda-12)/2} n}\right)^{i} \\
&\leq \tilde{O}(m)
\end{aligned}$$

Now consider the term involving m_d. We have that d is bounded by

$$\left(\frac{2}{3} + \delta\right) \log m / \log\left(\frac{1}{c_{PC}} \log n^{\lambda(\lambda-6)}\right).$$

Combining with the $\kappa_i = \log^{\lambda^2} n$, we get

$$\begin{aligned}
&\prod_{1 \leq j \leq d} \sqrt{\kappa(n_j)} \\
&= \left(\log n^{\lambda^2/2}\right)^{(\frac{2}{3}+\delta)\log m / \log(c \log n^{\lambda(\lambda-6)})} \\
&= \exp\left(\log\log n \frac{\lambda^2}{2}(\frac{2}{3}+\delta)\frac{\log m}{\lambda(\lambda-6)\log\log n - \log c_{PC}}\right) \\
&\leq \exp\left(\log\log n \frac{\lambda^2}{2}(\frac{2}{3}+\delta)\frac{\log m}{\lambda(\lambda-7)\log\log n}\right) \\
&\qquad (\text{since } \log c_{PC} \geq -\log n) \\
&= \exp\left(\log n \frac{\lambda}{\lambda-7}(\frac{1}{3}+\frac{\delta}{2})\right) \\
&= O(m^{(\frac{1}{3}+\frac{\delta}{2})\frac{\lambda}{\lambda-7}})
\end{aligned}$$

Since $m_d = O(m^{\frac{1}{3}-\delta})$, the total work is bounded by

$$O(m^{(\frac{1}{3}+\frac{\delta}{2})\frac{\lambda}{\lambda-7}+\frac{2}{3}-2\delta}) = O(m^{1+\frac{7}{\lambda-7}-\delta\frac{\lambda-14}{\lambda-7}})$$

So, setting $\delta \geq \frac{7}{\lambda-14}$ suffices to bound the total work by $\widetilde{O}(m)$. And, when δ is set to $\frac{7}{\lambda-14}$, the total parallel running time is bounded by the number of times the last layer is called

$$\begin{aligned}\prod_j \sqrt{\kappa(n_j)} \leq & O(m^{(\frac{1}{3}+\frac{1}{2(\lambda-14)})\frac{\lambda}{\lambda-7}}) \\ \leq & O(m^{\frac{1}{3}+\frac{7}{\lambda-14}+\frac{\lambda}{2(\lambda-14)(\lambda-7)}}) \\ \leq & O(m^{\frac{1}{3}+\frac{14}{\lambda-14}}) \quad \text{when } \lambda \geq 21\end{aligned}$$

Setting λ arbitrarily large suffices to give $O(m^{1/3+\theta})$ depth. ■

To match the promised bounds in Theorem 6.1, we improve the performance by reducing the exponent on the $\log n$ term in the total work from λ^2 to some large fixed constant while letting total depth still approach $O(m^{1/3+\theta})$.

Proof of Theorem 6.1: Consider setting $\lambda = 13$ and $\eta \geq \lambda$. Then,

$$\eta\lambda - 2\eta - 4\lambda \geq \eta(\lambda - 6) \geq \frac{7}{13}\eta\lambda$$

We use c_4 to denote this constant of $\frac{7}{13}$, namely c_4 satisfies

$$c_{PC}/\log^{\eta k-2\eta-4\lambda} n \leq c_{PC}/\log^{c_4\eta\lambda} n$$

We can then pick a constant threshold L and set κ_i for all $i \leq L$ as follows:

$$\kappa_1 = \log^{\lambda^2} n, \kappa_2 = \log^{(2c_4)\lambda^2} n, \cdots, \kappa_i = \log^{(2c_4)^{i-1}\lambda^2} n$$

To solve A_L, we apply Lemma 6.28, which is analogous to setting $A_L, \ldots, A_d$ uniformly. The depth required in constructing these preconditioners is $O(m_d + \sum_{j=1}^{L}(2c_4)^{j-1}\lambda^2)$, plus $O(m_d)$ for computing the inverse at the last level—for a total of $O(m_d) = O(m^{1/3})$.

As for work, the total work is bounded by

$$
\begin{aligned}
&\sum_{i\leq d} m_i \prod_{1\leq j\leq i} \sqrt{\kappa_j} + \prod_{1\leq j\leq d} \sqrt{\kappa_j} m_d^2 \\
&\quad = \sum_{i<L} m_i \prod_{1\leq j\leq i} \sqrt{\kappa_j} + \left(\prod_{1\leq j<L} \sqrt{\kappa_j}\right) \cdot \left(\sqrt{\kappa_j} \sum_{i\geq L} m_i \prod_{L\leq j\leq i} \sqrt{\kappa_j} + m_d^2 \prod_{L\leq j\leq d} \sqrt{\kappa_j}\right) \\
&\quad \leq \sum_{i<L} m_i \prod_{1\leq j\leq i} \sqrt{\kappa_j} + \left(\prod_{1\leq j<L} \sqrt{\kappa_j}\right) m_L \sqrt{\kappa_L} \\
&\quad = \sum_{i\leq L} m_i \prod_{1\leq j\leq i} \sqrt{\kappa_j} \\
&\quad \leq \sum_{i\leq L} \frac{m}{\prod_{j<i} \kappa_i^{c_4}} \prod_{1\leq j\leq i} \sqrt{\kappa_j} \\
&\quad = m \sum_{i\leq L} \frac{\sqrt{\kappa_1} \prod_{2\leq j\leq i} \sqrt{\kappa_{j-1}^{2c_4}}}{\prod_{j<i} \kappa_i^{c_4}} \\
&\quad = mL\sqrt{\kappa_1}
\end{aligned}
$$

The first inequality follows from the fact that the exponent of $\log^n$ in κ_L can be arbitrarily large, and then applying Lemma 6.28 to the solves after level L. The fact that $m_{i+1} \leq m_i \cdot O(1/\kappa_i^{c_4})$ follows from Lemma 6.21.

Since L is a constant, $\prod_{1\leq j\leq L} \in O(\operatorname{polylog} n)$, so the total depth is still bounded by $O(m^{1/3+\theta})$ by Lemma 6.28. ■

6.7 Conclusion

We presented a near linear-work parallel algorithm for constructing graph decompositions with strong-diameter guarantees and parallel algorithms for constructing spanning trees with stretch $2^{O(\sqrt{\log n \log\log n})}$ and ultra-sparse subgraphs with stretch $O(\log^{O(1)} n)$. The ultra-sparse subgraphs were shown to be useful in the design of a near linear-work parallel SDD solver. By plugging our result into previous frameworks, we obtained improved parallel algorithms for several problems on graphs.

Chapter 7

Probabilistic Tree Embeddings, k-Median, and Buy-at-Bulk Network Design

Probabilistic tree embeddings—the general idea of embedding finite metrics into a distribution of dominating trees while maintaining distances in expectation—has proved to be a very useful and general technique in the algorithmic study of metric spaces [Bar98]. Their study has far-reaching consequences to understanding finite metrics and developing approximation algorithms on them. An elegant optimal algorithm for such tree embeddings was given by Fakcharoenphol, Rao, and Talwar (FRT); it maintains distances to within $O(\log n)$ in expectation [FRT04]. In this chapter, we study the problem of computing such embeddings in parallel, and use these to obtain parallel approximation algorithms for k-median and buy-at-bulk network design.

A crucial design constraint is to ensure that the parallel work of our algorithms remains close to that of the sequential counterparts (a.k.a. *work efficiency*) while achieving small, preferably polylogarithmic, depth (parallel time). Work efficiency is important since it means an algorithm is useful regardless of whether we have a modest number of processors (*one* being the most modest) or a larger number. Such algorithms limit the amount of resources used and hence presumably the cost of the computation (e.g. in terms of the energy used, or the rental cost of a machine in the "cloud"). We will be less concerned with polylogarithmic factors in the depth since such a measure is typically not robust across models.

Summary of Results

We give a parallel algorithm to embed any n-point metric into a distribution of hierarchically well-sparated trees (HSTs) with $O(n^2 \log n)$ (randomized) work and $O(\log^2 n)$ depth, while providing the same distance-preserving guarantees as FRT [FRT04]. The main challenge arises in making sure the depth of the computation is polylogarithmic even when the resulting tree is highly imbalanced—in contrast, the FRT algorithm, as stated, works level by level. This imbalance can occur when the ratio between the maximum and minimum distances in the metric space is large. Our contribution lies in recognizing an alternative view of the FRT algorithm and developing an efficient algorithm to exploit it. In addition, our analysis also implies probabilistic embeddings into trees without Steiner nodes of height $O(\log n)$ whp. (though not HSTs); such trees are useful for both our algorithms and have also proved useful in other contexts.

Using this algorithm, we give an RNC $O(\log k)$-approximation for k-median. This is the first RNC algorithm that gives non-trivial approximation for any k [1]. Furthermore, the algorithm is work efficient relative to previously described sequential techniques. We also give an RNC $O(\log n)$-approximation algorithm for buy-at-bulk network design. This algorithm is within an $O(\log n)$ factor of being work efficient.

7.1 Preliminaries and Notation

For alphabet Σ and a sequence $\alpha \in \Sigma^*$, we denote by $|\alpha|$ the length of α and by α_i (or alternatively $\alpha(i)$) the i-th element of α. Given sequences α and β, we say that $\alpha \sqsubseteq \beta$ if α is a prefix of β. Furthermore, we denote by $\mathrm{LCP}(\alpha, \beta)$ the *longest common prefix* of α and β. Let $\mathsf{prefix}(\alpha, i)$ be the first i elements of α.

Let $G = (V, E)$ be a graph with edge lengths $\ell : E \to \mathbb{R}_+$. Let $d_G(u, v)$ or simply $d(u, v)$ denote the shortest-path distance in G between u and v. We represent graphs and trees in a form of adjacency array, where the vertices and the edges are each stored contiguously, and each vertex has a pointer to a contiguous array of pointers to its incident edges.

A *trie* (also known as a prefix tree) is an ordered tree where each tree edge is marked with a symbol from (constant-sized) Σ and a node v corresponds to the sequence given by the symbols on the root-to-v path, in that order. In this work, we only deal with non-empty sequences $s_1, s_2, \ldots, s_k$ of equal length. The trie corresponding to these sequences is one

[1]There is an RNC algorithm that give a $(5 + \varepsilon)$-approximation for $k \le \mathrm{polylog}(n)$ [BT10]

in which there are k leaf nodes, each corresponding uniquely to one of the sequences. If these sequences have long common prefixes, its trie has many long paths (i.e., a line of non-branching nodes). This can be compressed. Contracting all non-root degree-2 nodes by concatenating the symbols on the incident edges results in a *Patricia tree* (also known as a radix tree), in which by definition, all internal node except the root has degree at least 3. Using a (multiway-)Cartesian tree algorithm of Blelloch and Shun and known reduction [BS11], the Patricia tree of a lexicographically ordered sequence of strings $s_1, \ldots, s_n$ can be constructed in $O(n)$ work and $O(\log^2 n)$ depth assuming the following as input: (1) the sequences s_i's themselves, (2) $|s_i|$ the length of each s_i for $i \in [n]$, and (3) $|\mathrm{LCP}(s_i, s_{i+1})|$ the length of the longest common prefix between s_i and s_{i+1} for $i \in [n-1]$.

We also rely on the following primitives on trees. Given a commutative semigroup $(U, *)$, and a rooted tree T (not necessarily balanced) where every node $v \in V(T)$ is tagged with a value $\mathsf{val}(v) \in U$, there is an algorithm `treeAgg` that computes the aggregate value for each subtree of T (i.e., for each $v \in V(T)$, compute the result of applying $*$ to all $\mathsf{val}(\cdot)$ inside that subtree) in $O(n)$ work and $O(\log n)$ depth, assuming the binary operator $*$ is a constant-time operation [MRK88, JáJ92]. In the same work-depth bounds, the lowest common ancestor (LCA) of a pair of vertices u and v, denoted by $\mathrm{LCA}(u, v)$, can be determined (via the tree primitive just mentioned or otherwise) [SV88, BV93].

7.2 Parallel FRT Embedding

An input instance is a finite metric space (X, d), where $|X| = n$ and the symmetric distance function $d(\cdot, \cdot)$ is specified by an n-by-n matrix, normalized so that $1 \leq d(x, y) \leq 2^\delta = \Delta$ for all $x \neq y$. We adopt the standard convention that $d(x, x) = 0$.

In the sequential case, FRT [FRT04] developed an elegant algorithm that preserves the distances up to $O(\log n)$ in expectation. Their algorithm can be described as a top-down recursive *low-diameter decomposition* (LDD) of the metric. In broad strokes, given a metric space (X, d) with diameter Δ, the algorithm applies an LDD procedure to partition the points into clusters of diameter roughly $\Delta/2$, then each cluster into smaller clusters diameter of $\Delta/4$, etc. This construction produces a laminar family of clusters that we connect up based on set-inclusion, yielding a so-called FRT tree. The algorithm gives an optimal distance-preserving guarantee and can be implemented in $\widetilde{O}(n^2)$ sequential time.

The low-diameter decomposition step is readily parallelizable, but there is potentially a long chain of dependencies: for each i, figuring out the clusters with diameter 2^i requires knowl-

Algorithm 7.2.1 Implicit simultaneous low-diameter decompositions

```
1. Pick a permutation $\pi : X \to [n]$ uniformly at random.
2. Pick $\beta \in [1, 2]$ with the distribution $f_\beta(x) = 1/(x \ln 2)$.
3. For all $v \in X$, compute the partition sequence $\chi^{(v)}_{\pi,\beta}$.
```

edge of the clusters of diameter 2^{i+1}. This $O(\log \Delta)$ chain is undesirable for large Δ. Our algorithms get rid of this dependence. The main theorem of this section is the following:

Theorem 7.1 (Parallel FRT Embedding) *There is a randomized algorithm running in $O(n^2 \log n)$ work and $O(\log^2 n)$ depth that on input a finite metric space (X, d) with $|X| = n$, produces a tree T such that for all $x, y \in X$, $d(x, y) \le d_T(x, y)$ and $\mathbf{E}[d_T(x, y)] \le O(\log n)\, d(x, y)$.*

7.2.1 FRT Tree as Sequences

To achieve this parallelization, we take a slightly different, though completely equivalent, view of the FRT algorithm. (We assume basic familiarity with their algorithm.) Instead of a cluster-centric view which maintains a set of clusters that are refined over time, we explore a point-centric view which tracks the movement of each point across clusters but without explicitly representing the clusters. This view can also be seen as representing an FRT tree by the root-to-leaf paths of all external nodes (corresponding to points in the metric space). We formalize this idea in the following definition:

Definition 7.2 ((π, β)-Partition Sequence) *For $v \in X$, the* partition sequence *of v with respect to a permutation $\pi : X \to [n]$ and a parameter $\beta > 0$, denoted by $\chi^{(v)}_{\pi,\beta}$, is a length-$(\delta + 1)$ sequence such that $\chi^{(v)}_{\pi,\beta}(0) = 1$ and*

$$\chi^{(v)}_{\pi,\beta}(i) = \min\{\pi(w) \mid w \in X,\ d(v, w) \le \beta \cdot 2^{\delta-i-1}\}$$

for $i = 1, \ldots, \delta = \log \Delta$.

For each combination of π and β, this is the sequence of the lowest-numbered vertices (where the numbering is given by the random permutation π) that the node v can "see" as it reduces its range-of-vision geometrically from Δ down to 0—naturally, these numbers keep increasing from $1 = \min_{w\in X} \pi(w)$ to $\pi(v)$. Hence, the first step in generating an FRT tree is to pick a random permutation π on the nodes and a value β, and compute the partition sequence $\chi^{(v)}_{\beta,\pi}$ for each node $v \in X$, as in Algorithm 7.2.1. These partition sequences encode all the information we need to construct an FRT tree T as follows:

- *Vertices.* For $i = 0, \dots, \delta$, let $L_i = \{\mathsf{prefix}(\chi^{(v)}_{\pi,\beta}, i+1) \mid v \in X\}$ be the i-th level in the tree. The vertices of T are exactly $V(T) = \cup_i L_i$, where each $v \in X$ corresponds to the node identified by the sequence $\chi^{(v)}_{\pi,\beta}$.
- *Edges.* Edges only go between L_i and L_{i+1} for $i \geq 1$. In particular, a node $x \in L_i$ has an edge with length $2^{\delta-i}$ to $y \in L_{i+1}$ if $x \sqsubseteq y$.

We can check that this construction yields a tree because edges are between adjacent levels and defined by the subsequence relation. The following two lemmas show distance-preserving properties of the tree T. The first lemma indicates that d_T is an upper bound for d; the second shows that d_T preserves the distance d up to a $O(\log n)$ factor in expectation.

Lemma 7.3 *For all* $u, v \in X$*, for all* β, π*,* $d(u,v) \leq d_T(\chi^{(u)}_{\pi,\beta}, \chi^{(v)}_{\pi,\beta})$.

Lemma 7.4 *For all* $u, v \in X$*,* $\mathbf{E}\left[d_T(\chi^{(u)}_{\pi,\beta}, \chi^{(v)}_{\pi,\beta})\right] \leq O(\log n) \cdot d(u,v)$.

Since this is a completely equivalent process to that of [FRT04], the proofs are analogous but are given below for completeness.

Proof of Lemma 7.3: Let $u, v \in X$ such that $u \neq v$ be given. These nodes are "separated" at a vertex y that is the longest common prefix (LCP) of $\chi^{(u)}$ and $\chi^{(v)}$. Let $i^* = |\mathrm{LCP}(\chi^{(u)}, \chi^{(v)})|$. This means there is a vertex w at distance at most $\beta \cdot 2^{\delta - i^* - 1}$ from both u and v, so $d(u,v) \leq 2^{\delta - i^* + 1}$. On the other hand, both $\chi^{(u)}$ and $\chi^{(v)}$ are in the subtree rooted at y; therefore, $d_T(\chi^{(u)}, \chi^{(v)}) \geq 2 \cdot 2^{\delta - i^*} \geq 2^{\delta - i^* + 1}$, which concludes the proof. ∎

Before proving the upperbound on the tree distance, we state a useful fact: Because β is picked from $[1,2]$ with pdf. $\frac{1}{x \ln 2}$, the probability that for any $x \geq 1$, there exists an $i \in \mathbb{Z}_+ \setminus \{0\}$ such that $\beta \cdot 2^{i-1} \in [x, x+dx)$ is at most $\frac{dx}{x \ln 2}$:

Fact 7.5

$$\mathbf{Pr}\left[\exists i \geq 1, \beta \cdot 2^{i-1} \in [x, x+dx)\right] \leq \frac{dx}{x \ln 2}.$$

We now prove an upperbound on the distance:

Proof of Lemma 7.4: This proof is adapted from the original FRT proof to suit our setting. Let distinct $u, v \in X$ be given. For a level t, we say that $w \in X$ *settles* the pair uv if w is the first node (according to π) such that there exists an $x \in L_t$ with the property that $x_t = w$, and $x \sqsubseteq \chi^{(u)}$ or $x \sqsubseteq \chi^{(v)}$. We say that $w \in X$ *cuts* the pair uv if w settles it but for such an x, either $x \sqsubseteq \chi^{(u)}$ or $x \sqsubseteq \chi^{(v)}$—and not both.

We know that uv must be cut at some level, and if it is cut at level i, $d_T(u,v) \leq 2 \cdot \sum_{i' \geq i} 2^{\delta - i'} = 2^{\delta-i+2} \leq 8\beta 2^{\delta-i-1}$.

Let w be any node. Assume WLOG $d(w,u) \leq d(w,v)$. For w to cut uv, there must be a level i such that

1. $d(w,u) \leq \beta \cdot 2^{\delta-i-1} \leq d(w,v)$, and
2. w settles uv at this level i.

For the purpose of analysis, let's order the vertices of X as $w_1, w_2, \ldots$ in order of increasing distance from the closer of u and v. Consider the k-th node w_k. Like in FRT's proof, for a particular x between $d(w_k,u)$ and $d(w_k,v)$, $\mathbf{Pr}\left[\exists i \geq 1, \beta \cdot 2^{\delta-i-1} \in [x, x+dx)\right] \leq \frac{dx}{x \ln 2}$. Conditioned on this, at this level i, $w_1, w_2, \ldots, w_s$ could all settle uv, but the first in the π order will settle it. Thus, the contribution of this vertex to $d_T(u,v)$, in expectation, is at most

$$\int_{d(w_k,u)}^{d(w_k,v)} \frac{1}{x \ln 2} 8x \frac{1}{k} dx \leq O(\tfrac{1}{k} \cdot d(u,v)).$$

Summing over possible nodes that could cut uv, we have

$$\mathbf{E}\left[d_T(u,v)\right] \leq \sum_{i=1}^{n} O(\tfrac{1}{k} d(u,v)) = O(\log n) d(u,v).$$

■

7.2.2 A Simple Parallel FRT Algorithm

We now present a naïve parallelization of the above construction: not only does its depth depend on $\log \Delta$, it also does significantly more work than the sequential FRT algorithm. A parallel algorithm with such parameters can be inferred directly from [FRT04], but the presentation here is instructive: it relies on computing partition sequences and build the tree using them, a view that will be useful to get the improved parallel algorithm in the next section.

Lemma 7.6 *Given π and β, each $\chi_{\pi,\beta}^{(v)}$ can be computed in $O((n + \log \Delta) \log n)$ work and $O(\log n)$ depth.*

Proof: Let $v \in X$, together with π and β, be a given. We can sort the vertices by the distance from v so that $v = v_n$ and $d(v, v_1) \geq d(v, v_2) \geq \cdots \geq d(v, v_n) = 0$, where $v_1, \ldots, v_n$

are distinct vertices. This requires $O(n \log n)$ work and $O(\log n)$ depth. Then, we use a prefix computation to compute $\ell_i = \min\{\pi(v_j) \mid j \geq i\}$ for $i = 1, \ldots, n$. This quantity indicates that by going to distance at least $d(v, v_i)$, v could be a number as low as ℓ_i. This step requires $O(n)$ work and $O(\log n)$ depth (using a prefix computation). Finally, for each $k = 1, \ldots, \delta$, use a binary search to determine the smallest index i (i.e., largest distance) such that $d(v, v_i) \leq \beta \cdot 2^{\delta-k-1}$—and $\chi^{(v)}(k)$ is simply ℓ_i. There are $O(\log \Delta)$ such k values, each independently running in $O(\log n)$ depth and work, so this last step requires $O(\log \Delta \log n)$ work and $O(\log n)$ depth, which completes the proof. ■

Using this algorithm, we can compute all partition sequences independently in parallel, leading to a total of $O(n(n + \log \Delta) \log n)$ work and $O(\log n)$ depth for computing $\chi^{(v)}$ for all $v \in X$. The next step is to derive an embedding tree from these partition sequences. From the description in the previous section, to compute the set of level-i vertices, we examine all length-i prefixes $\mathsf{prefix}(\chi^{(v)}_{\pi,\beta}, i)$ for $v \in X$ and remove duplicates. The edges are easy to derive from the description. Each level i can be done in expected $O(i^2)$ work and $O(\log n)$ depth, so in total we need $O(\log^3 \Delta)$ work and $O(\log n \log \Delta)$ depth in expectation to build the tree from these sequences, proving the following proposition:

Proposition 7.7 *There is an algorithm* `simpleParFRT` *that computes an FRT tree in* $O(n^2 \log n + n \log \Delta \log n + \log^3 \Delta)$ *work and* $O(\log n \log \Delta)$ *depth.*

7.2.3 An Improved Algorithm

The simple algorithm in the preceding section had a $\log \Delta$ dependence in both work and depth. Now we show the power of the partition sequence view of the construction, and derive an algorithm whose work and depth bounds are independent of Δ. Moreover, the algorithm performs essentially the same amount of work as the sequential algorithm.

At first glance, the $\log \Delta$ dependence in the generation of partition sequences in our previous algorithm seems necessary and the reason is simple: the length of each partition sequence is $O(\log \Delta)$. To remove this dependence, we work with a different representation of partition sequences, one which has length at most n. This representation is based on the observation that any partition sequence is non-decreasing and its entries are numbers between 1 and n. Consequently, the sequence cannot change values more than n times and we only have to remember where it changes values. This inspires the following definition:

Definition 7.8 (Compressed Partition Sequence) *For $v \in X$, the* compressed partition sequence *of v, denoted by $\sigma^{(v)}_{\pi,\beta}$, is the unique sequence $\langle (s_i, p_i) \rangle_{i=1}^{k}$ such that* $1 = s_1 < \cdots <$

$s_k < s_{k+1} = \delta+1, p_1 < p_2 < \cdots < p_k$, and for all $i \leq k$ and $j \in \{s_i, s_i+1, \ldots, s_{i+1}-1\}$, $\chi^{(v)}_{\pi,\beta}(j) = p_i$, where $\chi^{(v)}_{\pi,\beta}$ is the partition sequence of v.

In words, if we view π as assigning priority values to X, then the compressed partition sequence of v tracks the distance scales at which the lowest-valued vertex within reach from v changes. As an example, at distances $\beta \cdot 2^{\delta-s_1+1}, \beta \cdot 2^{\delta-(s_1+1)+1}, \ldots, \beta \cdot 2^{\delta-(s_2-1)+1}$, the lowest-valued vertex within reach of v is p_1—and $\beta \cdot 2^{\delta-s_2+1}$ is the first distance scale at which p_1 cannot be reached and $p_2 > p_1$ becomes the new lowest-valued node. The following lemma shows how to efficiently compute the compressed partition sequence of a given vertex.

Lemma 7.9 *Given π and β, each compressed partition sequence $\sigma^{(v)}_{\pi,\beta}$ can be computed in $O(n \log n)$ work and $O(\log n)$ depth.*

Proof: The idea is similar to that of the partition sequence, except for how we derive the sequence at the end. Let $v \in X$, together with π and β, be a given. Sort the vertices by the distance from v so that $v = v_n$ and $d(v, v_1) \geq d(v, v_2) \geq \cdots \geq d(v, v_n)$, where $v_1, \ldots, v_n$ are distinct vertices. This has $O(n \log n)$ work and $O(\log n)$ depth. Again, compute $\ell_i = \min\{\pi(v_j) \mid j \geq i\}$ for $i = 1, \ldots, n$. Furthermore, let $b_i = \max\{j \geq 1 \mid \beta \cdot 2^{\delta-j-1} \geq d(v, v_i)\}$ for all $i = 1, \ldots, n$. This index b_i represents the smallest distance scale that v can still see v_i. Then, we compute $\rho_i = \min\{\ell_j \mid b_j = b_i\}$. Because the b_i's are non-decreasing, computing ρ_i's amounts to identifying where b_i's change values and performing a prefix computation. Thus, the sequences ℓ_i's, b_i's, and ρ_i's can be computed in $O(n)$ work and $O(\log n)$ depth.

To derive the compressed partition sequence, we look for all indices i such that $\rho_{i-1} \neq \rho_i$ and $b_{i-1} \neq b_i$—these are precisely the distance scales at which the current lowest-numbered vertex becomes unreachable from v. These indicies can be discovered in $O(n)$ work and $O(1)$ depth, and using standard techniques involving prefix sums, we put them next to each other in the desired format. ■

To keep the work term independent of $\log \Delta$, we cannot, for example, explicitly write out all the cluster nodes. The FRT tree has to be in a specific "compressed" format for the construction to be efficient. A *compacted FRT tree* is obtained by contracting all degree-2 internal nodes of an FRT tree, so that every internal node except for the root has degree at least 3 (a single parent and at least 2 children). By adding the weights of merged edges, the compacting preserves the distance between every pair of leaves. Equivalently, an FRT tree as described earlier is in fact a trie with the partition sequences as its input—and a compacted FRT tree is a *Patricia (or radix) tree on these partition sequences.*

Our task is therefore to construct a Patricia tree given compressed partition sequences. As discussed in Section 7.1, the Patricia tree of a *lexicographically ordered* sequence of strings $s_1, \ldots, s_n$ can be constructed in $O(n)$ work and $O(\log^2 n)$ depth if we have the following as input: (1) $|s_i|$ the length of each s_i for $i \in [n]$, and (2) $\mathrm{LCP}(s_i, s_{i+1})$ the length of the longest common prefix between s_i and s_{i+1} for $i \in [n-1]$. These sequences can be lexicographically ordered in $O(n^2 \log n)$ work and $O(\log^2 n)$ depth and the LCP between all adjacent pairs can be computed in $O(n^2)$ work and $O(\log n)$ depth. Combining this with the Patricia tree algorithm [BS11] gives the promised bounds, concluding the proof of Theorem 7.1.

7.2.4 FRT Tree Without Steiner Vertices

Some applications call for a tree embedding solution that consists of only the original input vertices. To this end, we describe how to convert a compacted FRT tree from the previous section into a tree that contains no Steiner vertices. As byproduct, the resulting non-Steiner tree has $O(\log n)$ depth with high probability.

Theorem 7.10 *There is an algorithm* `FRTNoSteiner` *running in* $O(n^2 \log n)$ *work and* $O(\log^2 n)$ *depth that on input an* n*-point metric space* (X, d)*, produces a tree* T *such that (1)* $V(T) = X$*; (2)* T *has* $O(\log n)$ *depth* ***whp.****; and (3) for all* $x, y \in X$*,* $d(x, y) \leq d_T(x, y)$ *and* $\mathbf{E}\left[d_T(x, y)\right] \leq O(\log n)\, d(x, y)$*.*

We begin by recalling that each leaf of an FRT tree corresponds to a node in the original metric space, so there is a bijection $f : L_\delta \to X$. Now consider an FRT tree T on which we will perform the following transformation: (1) obtain T' by multiply all the edge lengths of T by 2, (2) for each node $x \in V(T')$, label it with $\mathsf{label}(x) = \min\{\pi(f(y)) \mid y \in \mathsf{leaves}(T'_x)\}$, where $\mathsf{leaves}(T'_x)$ is the set of leaf nodes in the subtree of T' rooted at x, and (3) construct T'' from T' by setting an edge to length 0 if the endpoints are given the same label—otherwise retaining the length. As shown in Lemma 7.12, the resulting tree T'' has $O(\log n)$ depth with high probability.

Several things are clear from this transformation: First, for all $u, v \in X$, $d(u, v) \leq d_{T'}(\chi^{(u)}, \chi^{(v)})$ and $\mathbf{E}\left[d_{T'}(\chi^{(u)}, \chi^{(v)})\right] \leq O(\log n)\, d(u, v)$ (of course, with worse constants than d_T). Second, T'' is no longer an HST, but $d_{T''}$ is a lowerbound on $d_{T'}$, i.e., for all $u, v \in X$, $d_{T''}(\chi^{(u)}, \chi^{(v)}) \leq d_{T'}(\chi^{(u)}, \chi^{(v)})$. Therefore, to prove distance-preserving guarantees similar to Theorem 7.1, we only have to show that $d_{T''}$ dominates d

Lemma 7.11 *For all $u, v \in X$, $d(u,v) \leq d_{T''}(\chi^{(u)}, \chi^{(v)})$.*

Proof: Let $u \neq v \in X$ be given and let $y = \text{LCP}(\chi^{(u)}, \chi^{(v)})$. In T'' (also T and T'), y is the lowest common ancestor of $\chi^{(u)}$ and $\chi^{(v)}$. Let $i^* = |\text{LCP}(\chi^{(u)}, \chi^{(v)})|$. This means there is a vertex w at distance at most $\beta \cdot 2^{\delta - i^* - 1}$ from both u and v, so $d(u,v) \leq 2^{\delta - i^* + 1}$. Now let a (resp. b) be the child of y such that T_a (resp. T_b) contains $\chi^{(u)}$ (resp. $\chi^{(v)}$). So then, we argue that $d_{T''}(\chi^{(u)}, \chi^{(v)}) \geq 2 \cdot 2^{\delta - i^*}$ because $\mathsf{label}(y)$ must differ from at least one of the labels $\mathsf{label}(a)$ and $\mathsf{label}(b)$—and such non-zero edges have length $2^{\delta - i^* + 1}$ since we doubled its length in Step (1). This establishes the stated bound. ■

Lemma 7.12 *The depth of T'' is $O(\log n)$ with high probability.*

Proof: It is easy to see that the depth of T'' is upperbounded by the length of the longest compressed partition sequence. Consider a vertex $v \in X$ and let $v_1, \ldots, v_{n-1} \in X$ be such that $d(v, v_1) > d(v, v_2) > \cdots > d(v, v_{n-1}) > 0$. The length of the compressed partition sequence of v is upperbounded by the number of times the sequence $y_i = \min\{\pi(v_j) \mid n-1 \geq j \geq i\}$ changes value, a quantity which is known to be bounded by $O(\log n)$ with probability exceeding $1 - n^{-(c+1)}$ [Sei92]. Taking union bounds gives the desired lemma. ■

On a compacted FRT tree, this transformation is easy to perform. First, we identify all leaves with their corresponding original nodes. Computing the label for all nodes can be done in $O(n)$ work and $O(\log n)$ depth using `treeAgg` (Section 7.1), which is a variant of tree contraction. Finally, we just have to contract zero-length edges, which again, can be done in $O(n)$ work and $O(\log n)$ depth using standard techniques [JáJ92]. Note that we only have to compute the minimum on the nodes of a compacted tree, because the label (i.e., the minimum value) never changes unless the tree branches.

7.3 The k-Median Problem

The k-median problem is a standard clustering problem, which has received considerable attention in the past decades from various research communities. The input to this problem is a set of vertices $V \subseteq X$, where (X, d) is a (finite) metric space, and the goal is to find a set of at most k centers $F_S \subseteq V$ that minimizes the objective

$$\Phi(F_S) = \sum_{j \in V} d(j, F_S).$$

Since we will be working with many different metric spaces, we will write

$$\Phi_D(F_S) = \sum_{j\in V} D(j, F_S)$$

to emphasize which distance function is being used. In the sequential setting, several approximation algorithms are known, including $O(1)$-approximations (see [AGK$^+$04] and references therein) and approximation via tree embeddings [Bar98, FRT04]. In the parallel setting, these algorithms seem hard to parallelize directly: to our knowledge, the only RNC algorithm k-median gives a $(5+\varepsilon)$-approximation but only achieves polylogarithmic depth when k is at most polylog(n) [BT10].

Our goal is to obtain an $O(\log k)$-approximation that *has polylogarithmic depth for all k* and has essentially the same work bound as the sequential counterpart. The basic idea is to apply bottom-up dynamic programming to solve k-median on a tree, like in Bartal's paper [Bar98]. Later, we describe a sampling procedure to improve the approximation guarantee from $O(\log n)$ to $O(\log k)$. While dynamic programming was relatively straightforward to apply in the sequential setting, more care is needed in the parallel case: the height of a compacted FRT tree can be large, and since the dynamic program essentially considers tree vertices level by level, the total depth could be much larger than polylog(n).

Rather than working with compacted FRT trees, we will be using FRT trees that contain no Steiner node, constructed by the algorithm `FRTNoSteiner` in Theorem 7.10. This type of trees is shown to have the same distance-preserving properties as an FRT tree but has $O(\log n)$ depth with high probability. Alternatively, we give an algorithm that reduces the depth of a compacted FRT tree to $O(\log n)$; this construction, which assumes the HST property, is presented in Section 7.6.3 and may be of independent interest.

7.3.1 Solving k-Median on Trees

Our second ingredient is a parallel algorithm for solving k-median when the distance metric is the shortest-path distance in a (shallow) tree. For this, we will parallelize a dynamic programming (DP) algorithm of Tamir [Tam96], which we now sketch. Tamir presented a $O(kn^2)$ algorithm for a slight generalization of k-median on trees, where in his setting, each node $i \in V$ is associated with a cost c_i if it were to be chosen; every node i also comes equipped with a nondecreasing function f_i; and the goal becomes to find a set $A \subseteq V$ of size at most k to minimize

$$\sum_{i\in A} c_i + \sum_{j\in V} \min_{i\in A} f_j(d(v_j, v_i)).$$

This generalization (which also generalizes the facility-location problem) provides a convenient way of taking care of Steiner nodes in FRT trees. For our purpose, these f_i's will simply be the identity function[2] $x \mapsto x$, and c_i's are set so that it is 0 if i is a real node from the original instance and ∞ if i is a Steiner node (to prevent it from being chosen).

Tamir's algorithm is a DP which solves the problem exactly, and for simplicity, it requires the input tree to be binary. Tamir also gave an algorithm that converts any tree into a binary tree. His algorithm, however, can significantly increase the depth. For our algorithm, *we need a different an algorithm that ensures not only that the tree is binary but also that the depth does not grow substantially.* To this end, we give a parallel algorithm based on the Shannon-Fano code construction [Sha48] in Section 7.6.1, which outputs a binary tree whose depth is an additive $\log n$ larger than the original depth. We set edge lengths as follows: The new edges will have length 0 except for the edges incident to the original v_i's: the parent edge incident to v_i will inherit the edge length from $v_i v$. Also, the added nodes have cost ∞. As a result of this transformation, the depth of the new tree is at most $O(\log n)$ and the number of nodes will at most double. Furthermore, the whole transformation can be accomplished in $O(\log^2 n)$ depth and $O(n \log n)$ work (see Section 7.6.1).

The main body of Tamir's algorithm begins by producing for each node v a sequence of vertices that orders all vertices by their distances from v with ties appropriately broken. His algorithm for generating these sequences are readily parallelizable because the sequence for a vertex v involves merging the sequences of its children in a manner similar to the merge step in merge sort. The work for this step, as originally analyzed, is $O(n^2)$. Each merge can be done in $O(\log n)$ depth and there are at most $O(\log n)$ levels; this step has depth $O(\log^2 n)$.

Armed with this, the actual DP is straightforward to parallelize. Tamir's algorithm maintains two tables F and G, both indexed by a tuple (i, q, r) where $i \in [n]$ represents a node, $q \leq k$ counts the number of centers inside the subtree rooted at i, and r indicates roughly the distance from i to the closest selected center outside of the subtree rooted at i. As such, for each i and q, there can be at most n different values for r. Now Tamir's DP is amendable to parallelization because the rules of the DP compute an entry using *only* the values of its immediate children. Further, each rule is essentially taking the minimum over a combination of parameters and can be parallelized using standard algorithms for finding the minimum and prefix sums. Therefore, we can compute the table entries for each level of the tree in general and move on to the higher level. It is easy to check each level can be accomplished in $O(\log n)$ depth, and as analyzed in Tamir's paper, the total work is bounded by $O(kn^2)$.

[2]In the weighted case, we will use $f_i(x) = w_i \cdot x$ to reflect the weight on node i.

7.3.2 Parallel Successive Sampling

Our algorithm thus far gives an $O(\log n)$-approximation on input consisting of n points. To improve this to $O(\log k)$, we devise a parallel version of Mettu and Plaxton's successive sampling (shown in Algorithm 7.3.1) [MP04]. We then describe how to apply it to our problem. Since the parallel version produces an identical output to the sequential one, guarantees about the output follow directly from the results of Mettu and Plaxton. Specifically, they showed that there are suitable settings of α and β such that by using $K = \max\{k, \log n\}$, the algorithm runs for $O(\log(n/K))$ rounds and produces Q of size at most $O(K \cdot \log(n/K))$ with the following properties:

Theorem 7.13 (Mettu and Plaxton [MP04]) *There exists an absolute constant C_{SS} such that if $Q = \texttt{SuccSampling}(V, K)$, then with high probability, Q satisfies $\Phi(Q) \leq C_{SS} \cdot \Phi(OPT_k)$, where OPT_k is an optimal k-median solution on the instance V.*

In other words, the theorem says that Q is a bicriteria approximation which uses $O(K \log(n/K))$ centers and obtains a C_{SS}-approximation to k-median. To obtain a parallel implementation of successive sampling (Algorithm 7.3.1), we will make steps 1–3 parallel. We have the following runtime bounds:

Lemma 7.14 *For input V with $|V| = n$ and $K \leq n$, $\texttt{SuccSampling}(V, K)$ has $O(nK)$ work and $O(\log^2 n)$ depth.*

Proof: First, by the choice of C_i, we remove at least a β fraction of U_i and since α and β are constants, we know that the number of iterations of the **while** loop is $O(\log(n/K))$. Now step 1 of the algorithm can be done in $O(nK)$ work and $O(1)$ depth (assuming concurrent writes). To perform Step 2, First, we compute for each $p \in U_i$, the distance to the nearest point in S_i. This takes $O(|U_i|K)$ work and $O(\log K)$ depth. Then, using a linear-work selection algorithm, we can find the set C_i and r_i in $O(|U_i|)$ work and $O(\log |U_i|)$ depth. Since each time $|U_i|$ shrinks by a factor β, the total work is $O(nK)$ and the total depth is $O(\log^2 n)$, as claimed. ∎

Piecing together the components developed so far, we obtain a $O(\log k)$-approximation. The following theorem summarizes our main result for the k-median problem:

Theorem 7.15 *For $k \geq \log n$, the k-median problem admits a factor-$O(\log k)$ approximation with $O(nk + k(k \log(\frac{n}{k}))^2) \leq O(kn^2)$ work and $O(\log^2 n)$ depth. For $k < \log n$,*

Algorithm 7.3.1 `SuccSampling`$_{\alpha,\beta}(V, K)$—successive sampling

```
Let $U_0 = V$, $i = 0$
while ($|U_i| > \alpha K$)
    1. Sample from $U_i$ u.a.r. (with replacement) $\lfloor \alpha K \rfloor$ times—call the chosen points $S_i$.
    2. Compute the smallest $r_i$ such that $|B_{U_i}(S_i, r_i)| \geq \beta \cdot |U_i|$ and let $C_i = B_{U_i}(S_i, r_i)$,
       where $B_U(S, r) = \{w \in U \mid d(w, S) \leq r\}$.
    3. Let $U_{i+1} = U_i \setminus C_i$ and $i = i + 1$.
Output $Q = S_0 \cup S_1 \cup \cdots \cup S_{i-1} \cup U_i$
```

the problem admits a $O(1)$-approximation with $O(n \log n + k^2 \log^5 n)$ work and $O(\log^2 n)$ depth.

Here is a proof sketch, see Section 7.6.2 for more details: we first apply Algorithm 7.3.1 to get the set Q (in $O(nK)$ work and $O(\log^2 n)$ depth). Then, we "snap" the clients to their closest centers in Q (paying at most $C_{SS}\,\Phi(\mathrm{OPT}_k)$ for this), and depending on the range of k, either use an existing parallel k-median algorithm for $k < \log n$ [BT10] or use the FRT-based algorithm on these "moved" clients to get the $O(\log q)$-approximation (in $O(kq^2)$ work and $O(\log^2 q)$ depth, where $q = O(K \log(n/K))$, because we are running the algorithm only on $O(K \log(n/K))$ points). Note that we now need a version of the k-median algorithm on trees (Section 7.3.1) where clients also have weights, but this is easy to do (by changing the f_i's to reflect the weights).

7.4 Buy-at-Bulk Network Design

Let $G = (V, E)$ be an undirected graph with n nodes; edge lengths $\ell\colon E \to \mathbb{R}_+$; a set of k demand pairs $\{\mathsf{dem}_{s_i,t_i}\}_{i=1}^{k}$; and a set of cables, where cable of type i has capacity u_i and costs c_i per unit length. The goal of the problem is to find the cheapest set of cables that satisfy the capacity requirements and connect each pair of demands by a path. Awerbuch and Azar [AA97] gave a $O(\log n)$-approximation algorthim in the sequential setting. Their algorithm essentially finds an embedding of the shortest-path metric on G into a distribution of trees *with no Steiner nodes*, a property which they exploit when assigning each tree edge to a path in the input graph. For an edge with net demand dem, the algorithm chooses the cable type that minimizes $c_i\lceil \mathsf{dem}/u_i \rceil$. This is shown to be an $O(\log n)$-approximation. From a closer inspection, their algorithm can be parallelized by developing parallel algorithms for the following:

1. Given a graph G with edge lengths $\ell : E(G) \to \mathbb{R}_+$, compute a dominating tree T with no Steiner node such that T $O(\log n)$-approximate d in expectation.
2. For each $(u, v) \in E(T)$, derive the shortest path between u and v in G.
3. For each $e \in E(T)$, compute the net demand that uses this edge, i.e.,

$$f_e = \sum_{i:e \in P_T(s_i,t_i)} \mathsf{dem}_{s_i,t_i},$$

   where $P_T(u, v)$ denotes the unique path between u and v in T.

We consider these in turn. First, the shortest-path metric d can be computed using a standard all-pair shortest paths algorithm in $O(n^3 \log n)$ work and $O(\log^2 n)$ depth. With this, we apply the algorithm in Section 7.2.4, yielding a dominating tree T in which d_T $O(\log n)$-approximates d. Furthermore, for each tree $e = (u, v)$, the shortest-path between u and v can be recovered from the calculation performed to derive d at no additional cost.

Next we describe how to calculate the net demand on every tree edge. We give a simple parallel algorithm using the `treeAgg` primitive discussed in Section 7.1. As a first step, we identify for each pair of demands its least common ancestor (LCA), where we let LCA(u, v) be the LCA of u and v. This can be done in $O(n)$ work and $O(\log n)$ depth for each pair. Thus, we can compute the LCA for all demand pairs in $O(kn)$ work and $O(\log n)$ depth. As input to the second round, we maintain a variable $\mathsf{up}(w)$ for each node w of the tree. Then, for every demand pair $\mathsf{dem}_{u,v}$, we add to both $\mathsf{up}(u)$ and $\mathsf{up}(v)$ the amount of $\mathsf{dem}_{u,v}$—and to $\mathsf{up}(\text{LCA}(u, v))$ the *negative* amount $-2\mathsf{dem}_{u,v}$. As such, the sum of all the values up inside a subtree rooted at u is the amount of "upward" flow on the edge out of u toward the root. This is also the net demand on this edge. Therefore, the demand f_e's for all $e \in E(T)$ can be computed in $O(kn)$ work and $O(\log n)$ depth. Finally, mapping these back to G and figuring out the cable type (i.e., computing $\min_i c_i \lceil \mathsf{dem}/u_i \rceil$) are straightforward and no more expensive than computing the all-pair shortest paths. Hence, we have the following theorem:

Theorem 7.16 *The buy-at-bulk network design problem with k demand pairs on an n-node graph can be solved in $O(n^3 \log n)$ work and $O(\log^2 n)$ depth.*

7.5 Conclusion

We gave an efficient parallel algorithm for tree embedding with expected $O(\log n)$ stretch. Our contribution is in making these bounds independent of the ratio of the smallest to largest distance, by recognizing an alternative view of the FRT algorithm and developing an efficient algorithm to exploit it. As applications, we developed the first an RNC $O(\log k)$-approximation algorithm for k-median and an RNC $O(\log n)$-approximation for buy-at-bulk network design.

7.6 Appendix: Various Proofs

7.6.1 Making Trees Binary using Shannon-Fano Coding

Given a tree T with n nodes and depth h but of arbitrary fanout, we give a construction that yields a tree with depth $O(h + \log n)$. This construction makes use of the Shannon-Fano code's construction [Sha48], For each original tree node v with children $v_1, \dots, v_k$, we assign to v_i the probability

$$p_i = \frac{|V(T_{v_i})|}{|V(T_v)|},$$

where T_u denotes the subtree of T rooted at u. Shannon-Fano's result indicates that there is a binary tree (though, originally stated in terms of binary strings) whose external nodes are exactly these $v_1, \dots, v_k$ and the path from v to v_i in the new tree has length at most $1 + \log(1/p_i)$. Applying this construction on all internal nodes with degree more than 2 gives the following lemma:

Lemma 7.17 *Given a tree T with n nodes and depth h but of arbitrary fanout, there is an algorithm `treeBinarize` producing a binary tree with depth at most $h + \log n$.*

Proof: Let w be a leaf node in T and consider the path from the root node to w. Suppose on this path, the subtrees have sizes $n = n_1 > n_2 > \cdots > n_{d'} = 1$, where $d' \leq h$. Applying the aforementioned construction, this path expands to a path of length at most

$$1 + \sum_{i=1}^{d'-1} (1 + \log(1/p_i)) \;=\; 1 + \sum_{i=1}^{d'-1} (1 + \log(\tfrac{n_i}{n_{i+1}})) \;\leq\; d' + \log n \leq h + \log n,$$

which proves the lemma. ■

This construction is also easily parallelized as all that is needed is sorting the probabilities in decreasing order (so that $p_1 \geq p_2 \geq \cdots p_k$), computing the cumulative probabilities (i.e., $P_i = \sum_{i' \leq i} p_i$), finding the point where the cumulative probability splits in (roughly) half, and recursing on the two sides. Sorting needs to be done once for each v; then, each call involves essentially a prefix computation. Thus, the transformation on the whole tree will take $O(n \log n)$ work and $O(\log^2 n)$ depth. (Note that the degrees of T's nodes sum to at most $2n$.)

7.6.2 Piecing Together the k-median Algorithm

Case I: $k \geq \log n$:

Let $Q = \{q_1, \dots, q_K\} = \texttt{SuccSampling}(V, K)$, where $K = \max\{k, \log n\}$ and $\varphi : V \to Q$ be the mapping that sends each $v \in V$ to the closest point in Q (breaking ties arbitrarily). Thus, $\varphi^{-1}(q_1), \varphi^{-1}(q_2), \dots, \varphi^{-1}(q_K)$ form a collection of K non-intersecting clusters that partition V. For $i = 1, \dots, K$, we define $w(q_i) = |\varphi^{-1}(q_i)|$. We prove a lemma that relates a solution's cost in Q to the cost in the original space (V, d).

Lemma 7.18 *Let $A \subseteq Q \subseteq V$ be a set of k centers satisfying*

$$\sum_{i=1}^{K} w(q_i) \cdot d(q_i, A) \quad \leq \quad \beta \cdot \min_{\substack{X \subseteq Q \\ |X| \leq k}} \sum_{i=1}^{K} w(q_i) \cdot d(q_i, X)$$

for some $\beta \geq 1$. Then,

$$\Phi(A) = \sum_{x \in V} d(x, A) \leq O(\beta \cdot c_{SS}) \cdot \Phi(OPT_k),$$

where OPT_k, as defined earlier, is an optimal k-median solution on V.

Proof: For convenience, let

$$\lambda = \sum_{x \in V} d(x, Q) = \sum_{x \in V} d(x, \varphi(x)),$$

and so $\lambda \leq c_{\text{SS}} \cdot \Phi(\text{OPT}_k)$. We establish the following:

$$\begin{aligned}\sum_{x\in V} d(x,A) &\leq \lambda + \sum_{i=1}^{K} w(q_i)\cdot d(q_i,A) \leq \lambda + \beta \cdot \min_{\substack{X\subseteq Q\\ |X|\leq k}} \sum_{i=1}^{K} w(q_i)\cdot d(q_i,X)\\ &\leq \lambda + \beta\sum_{i=1}^{K} w(q_i)\cdot d(q_i,\varphi(\text{OPT}_k)) \leq \lambda + 2\beta\sum_{i=1}^{K} w(q_i)\cdot d(q_i,\text{OPT}_k)\\ &\leq \lambda + 2\beta\sum_{i=1}^{K}\sum_{x\in\varphi^{-1}(q_i)}\Big(d(q_i,x)+d(x,\text{OPT}_k)\Big)\\ &\leq \lambda + 2\beta\lambda + 2\beta\Phi(\text{OPT}_k) = O(\beta\cdot c_{\text{SS}})\cdot\Phi(\text{OPT}_k),\end{aligned}$$

which proves the lemma. ■

By this lemma, the fact that the k-median on tree gives a $O(\log|Q|)$-approximation, and the observation that $|Q| \leq k^2$ (because $k \geq \log n$), we have that our approximation is $O(\log k)$.

Case II: $k < \log n$:

We run successive sampling as before, but this time, we will use the parallel local-search algorithm [BT10] instead. On input consisting of n points, the BT algorithm has $O(k^2 n^2 \log n)$ work and $O(\log^2 n)$ depth. Since $k < \log n$, we have $K = \log n$ and the successive sampling algorithm would give $|Q| \leq \log^2 n$. This means we have an algorithm with total work $O(n\log n + k^2\log^5 n)$ and depth $O(\log^2 n)$.

7.6.3 Tree Trimming for k-Median

Another way to control the height of the tree is by taking advantage of a cruder solution (which can be computed inexpensively) to prune the tree. The first observation is that if $A \subseteq V$ is a ρ-approximation to k-center, then A is a ρn-approximation to k-median. Using a parallel k-center algorithm [BT10], we can find a 2-approximation to k-center in $O(n\log^2 n)$ work and $O(\log^2 n)$ depth. This means that we can compute a value β such that if OPT is an optimal k-median solution, then $\Phi(\text{OPT}) \leq \beta \leq 2n\cdot\Phi(\text{OPT})$.

Following this observation, two things are immediate when we consider an *uncompacted* FRT tree:

1. If we are aiming for a $C \cdot \log n$ approximation, no clients could go to distance more than than $C \cdot \log n \cdot \Phi(\text{OPT}) \leq C \cdot n\beta$. This shows twe can remove the top portion of the tree where the edge lengths are more than $Cn\beta$.
2. If a tree edge is shorter than $\tau = \frac{\beta}{8n^2}$, we could set its length to 0 without significantly affecting the solution's quality. These 0-length edges can be contracted together. Because an FRT tree as constructed in Theorem 7.1 is a 2-HST, it can be shown that if T' is obtained from an FRT tree T by the contraction process described, then for all $x \neq y$, $d_T(x, y) \leq d_{T'}(x, y) + 4\tau$.

Both transformations can be performed on compacted trees in $O(n)$ work and $O(\log n)$ depth, and the resulting tree will have height at most $O(\log(8n^3)) = O(\log n)$. Furthermore, if $A \subseteq V$ is any k-median solution, then

$$\begin{aligned}\Phi_d(A) \leq \Phi_{d_T}(A) = \sum_{x \in V} d_T(x, A) &\leq \sum_{x \in V} \left(d_{T'}(x, A) + 4t\right) \\ &\leq \Phi_{d_{T'}}(A) + n \cdot \frac{\beta}{2n^2} \leq \Phi_{d_{T'}}(A) + \Phi_d(\text{OPT}).\end{aligned}$$

Chapter 8

Hierarchical Diagonal Blocking

Iterative algorithms are often the method of choice for large sparse problems. For example, specialized multigrid linear solvers have been developed to solve large classes of symmetric positive definite matrices [BHM00, TSO00]. Many of these solvers run in near linear time and are being applied to very large systems. These algorithms heavily rely on the sparse matrix-vector multiplication (SpMV) kernel, which dominates the running time. As noted by many, the performance of SpMV on large matrices, however, is almost always limited by memory bandwidth. This is even more pronounced on modern multicore hardware where the aggregate memory bandwidth can be particularly limiting [WOV$^+$07b] when all the cores are busy.

Many approaches have been suggested to reduce the memory bandwidth requirements in SpMV: row/column reordering [PF90, OLHB02], register blocking [Tol97], compressing row or column indices [WL06] , cache blocking [IYV04, WOV$^+$07b], symmetry [Saa90], using single or mixed precision [BDK$^+$08], and reorganizing the SpMV ordering across multiple iterations in a solver [MHDY09], among others. Some of these approaches are hard to parallelize. For example, the standard sparse skyline format for symmetric matrices does not parallelize well.

The work in this chapter introduces an approach we refer to as *hierarchical diagonal blocking* (HDB) which we believe captures many of the existing optimization techniques in a common representation. It can take advantage of symmetry while still being easy to parallelize. It takes advantage of reordering. It also allows for simple compression of column indices. In conjunction with precision reduction (storing single-precision numbers in place of doubles), it can reduce the overall bandwidth requirements by more than a factor of three. It is particularly well-suited for the type of problems that CMG is designed for, symmetric matrices for which the corresponding graphs have reasonably small graph separators, and for which the effects of reduced precision arithmetic are well-understood. Our approach does not use register blocking although this could be added.

We prove various theoretical bounds for matrices for which the adjacency structure has edge separators of size $O(n^\alpha)$ for $\alpha < 1$. Prior work has shown a wide variety of sparse matrices have a graph structure with good separators [BBK04]. We study the algorithm in the cache-oblivious framework [FLPR99], where algorithms are analyzed assuming a two-level memory hierarchy with an unbounded main memory and a cache of size M and line size B. As long as the algorithm does not make use of any cache parameters, the bounds are simultaneously valid across all cache levels in a hierarchical cache. For an $n \times n$ matrix with m nonzeros, we show that the number of misses is at most $m/B + O(1 + n/(Bw) + n/M^{1-\alpha})$, where w is the number of bits in a word.

We complement the theoretical results with a number of experiments, evaluating the performance of various SpMV schemes on recent multicore architectures. Our results show that a simple double-precision parallel SpMV algorithm saturates the multicore bandwidth, but by reducing the bandwidth requirements—using a combination of hierarchical diagonal blocking and precision reduction—we are able to obtain, on average, a factor of 2.5x speedup on an 8-core Nehalem machine. We also examine the implications of using the improved SpMV routine in CMG and preconditioned conjugate gradient (PCG) solvers. In addition, we explore heuristics for finding good separator-orderings and study the effects of separator quality on SpMV performance.

Reducing SpMV Bandwidth Requirements

Prior work has proposed several approaches for reducing the memory bandwidth requirements of SpMV. Reordering of rows and columns of the matrix can reduce the cache misses on the input and output vectors x and y by bringing references to these vectors closer to each other in time [OLHB02]. Many heuristic reordering approaches have been used, including graph separators such as Chaco [HL95] or METIS [KK98], Cuthill-McKee reordering [GL81] or the Dulmage-Mendelsohn permutation [PF90]. These techniques tend to work well in practice since real-world matrices tend to have high locality. This is especially true with meshes derived from 2- and 3-d embeddings. Recent results have shown various bounds for meshes with good separators [BKTW07, BCG$^+$08, BGS10]. The graph structure of a wide variety of sparse matrices has been to shown to have good separators, including graphs such as the Google link graph. Reordering can be used with cache blocking [IYV04], which blocks the matrix into sparse rectangular blocks and processes each block separately so that the same rows and columns are reused.

Index compression reduces the size of the column and row indices used to represent the matrix. The indices are normally represented as integers, but there are various ways to reduce their size. Willcok and Lumsdaine [WL06] apply graph compression techniques to reduce the size, showing speedups of up to 33% (although much more modest numbers on average). Williams et al. point out that by using cache blocking, it is possible to reduce the number of bits for the column indices since the number of columns in the block is typically small [WOV+07b]. Register blocking [Tol97] represents the matrix as a set of dense blocks. This can reduce the index information needed, but for very sparse or unstructured matrices, it can cause significant fill due to the insertion of zero entries to fill the dense blocks.

Data compression is a natural extension of index compression that attempts to reduce the size of the actual data contained in the matrix. For symmetric matrices, one can store the lower-triangular entries and use them twice. When stored in the sparse skyline format [Saa90], (the compressed sparse row format with only elements strictly below the diagonal stored) a simple loop of the following form can be used:

```
// loop over rows.
for (i = 0;i < n;i++) {
  float sum = diagonal[i]*x[i];
  // loop over nonzeros below diagonal in row
  for (j = start[i];j < start[i+1];j++) {
    sum += x[cols[j]] * vals[j]; // as row
    y[cols[j]] += x[i] * vals[j]; // as column
  }
  y[i] += sum;
}
```

Figure 8.1: Simple sequential code for sparse matrix vector multiply (SpMV).

Unfortunately, this loop does not parallelize well because of the unstructured addition to an element in the result vector in the statement `y[cols[j]] += x[i] * vals[j];`. Buluç et al. study how to parallelize this by recursively blocking the matrix [BFF+09], but this does not take advantage of any locality in the matrix.

Another approach to data compression is to reduce the number of bits used by the nonzero entries. Buttari et al. [BDK+08] suggest the implementation of mixed-precision inner-outer

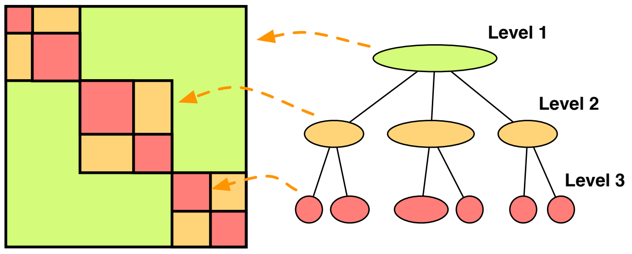


Figure 8.2: Hierarchical diagonal blocking: decomposing a matrix into a tree of submatrices.

iterative algorithms, i.e. a nesting of iterative algorithms where the outer iterative method is implemented in double precision, and the inner one—formally viewed as a preconditioner to the outer one—is implemented in single precision. While often positive, the effects of reduced precision are in general unpredictable. One main advantage of the CMG solver comparing to other iterative methods is that it can be used as a preconditioner to Conjugate Gradient, and the effects of using single precision are well-understood.

Finally, recent work by Mohiyuddin et al. [MHDY09] suggests reorganizing a sequence of SpMV operations on the same matrix structure across iterations so that the same part of the vector can be reused. Although this works well when using the same matrix over multiple iterations, it does not directly help in algorithms such as multigrid, where only a single iteration on a matrix is applied before moving to another matrix of quite different form.

8.1 Hierarchical Diagonal Blocking SpMV

In this section, we describe the *hierarchical diagonal blocking* (HDB) representation for sparse square matrices and an SpMV routine for the representation. We assume that we have already computed a fully balanced tree of edge-separators for the graph of the matrix, with the vertices as leaves. In the following discussion, we assume the rows of the square matrix are ordered by left-to-right pass over the leaves (the separator ordering), and since the matrix is square, we will use row to refer to both the row and corresponding column.

The *HDB representation* is a partitioning of the matrix into a tree of submatrices (see Figure 8.2). Each leaf represents a range of rows (possibly a single row), and each internal node of the tree represents a continuous range of rows it covers. Nonzero entries of the matrix are stored at the least common ancestor of the leaves containing its two indices (row and column). If both indices are in the same leaf, then the element will be stored at that leaf (all diagonal entries are at a leaf). The representation stores with each internal node the range of rows it covers, and we refer to the number of rows in the range as the node's size.

The separator tree can be used directly as the structure of the HDB tree. This, however, creates many levels which help neither in theory nor in practice. Instead, we coalesce the nodes of the separator tree so that sizes square at each level: $2, 4, 16, 256, 65536, \ldots, 2^{2^i}$. We maintain the separator ordering among the children of a node. This is important for the cache analysis. We note that for matrices with good separators most of the entries will be near the leaves.

Algorithm 8.1.1 Sparse Matrix Vector Multiply for HDB

```
HDB_SpMV ($x, y, T$):
 1: $A = T.M$ // the nonzero entries in this node of $T$
 2: $[\ell, u] = T.range$
 3: if isLeaf($T$) then
 4:     $y[\ell, u] = A \cdot x[\ell, u]$
 5: else
 6:     for all $t \in T$.children, in parallel, do
 7:         HDB_SpMV ($x, y, t$)
 8:     end for
 9:     $y[\ell, u] = y[\ell, u] + A \cdot x[\ell, u]$
10: end if
```

The SpMV routine on the HDB representation works as shown in Algorithm 8.1.1. The recursive algorithm takes as arguments the input vector x, the output vector y, and a subtree/internal node T. The algorithm requires that the ordering of the x and y vectors coincide with the separator (matrix) ordering. We denote by $[\ell, u]$ the range of rows the subtree covers. The algorithm computes the contribution to y for-all nonzero entries in the subtree. In the base case, it directly calculates the contribution. In the inductive case, each recursive call in the **for all** loop computes the contribution for the entries in its subtree. Since each of these is on a distinct range of rows, all the calls can be made in parallel without interference. After returning from the recursive calls, the algorithm adds in the contribution for the entries in the current node, hence accounting for the contribution of all entries in the subtree.

An important feature of HDB is that it gives freedom in the selection of the matrix representation A and corresponding SpMV algorithm used for each node of the tree. In particular, depending on the level, different representations can be used. If A in some node has many empty rows in its range, we need store only the non-empty rows. This can easily be done using an additional index vector of non-empty rows as is often done in cache-blocked algorithms [WOV+07b]. If the matrix is symmetric, then we can keep just the lower triangular part and store it in Compressed Sparse Row (CSR) format. For a submatrix stored in this form, we can use the skyline algorithm given in Figure 8.1 and for internal nodes, we need not even worry about diagonals. Since the skyline algorithm is difficult to parallelize, it can be used sequentially at lower levels of the tree where there is plenty of parallelism from the recursive calls, and the CSR representation with redundant entries can be used at the higher levels. This works both in theory (proof of Theorem 8.1) and in practice (Section 8.3). Another important feature of HDB is that space can be saved in storing the indices by only storing an offset relative to the beginning of the range. Again, this is used both in theory (Theorem 8.1) and practice (Section 8.3).

We now bound space, cache complexity, and depth for HDB_SpMV for matrices with good separators. We assume that each nonzero value takes one word of memory. Therefore, B nonzeros fit in a cache line (this is just the values and not any indices). We assume a word has w bits in it.

Theorem 8.1 *Let $\mathcal{M}$ be a class of matrices for which the adjacency graphs satisfy an n^α-edge separator theorem, $\alpha < 1$, and $A \in \mathcal{M}$ be an $n \times n$ matrix with $m \geq n$ nonzeros, or $m \geq n$ lower triangular nonzeros for a symmetric matrix. If A is stored in the HDB representation T then:*

1. *T can be implemented to use $m + O(n/w)$ words.*
2. *Algorithm HDB_SpMV (x, y, T) is cache oblivious and runs with $m/B + O(1 + n/(Bw) + n/M^{1-\alpha})$ misses in the ideal cache model.*
3. *Algorithm HDB_SpMV (x, y, T) runs in $O(\log^c n)$ depth (span) for some constant c.*

The proof of this theorem relies on the follong result for sparse-matrix vector matrix multiply.

Theorem 8.2 (Blelloch et al. [BCG+08]) *Let $\mathcal{M}$ be a class of matrices for which the adjacency graphs satisfy an n^α-edge separator theorem with $\alpha < 1$. Any $n \times n$ matrix $A \in \mathcal{M}$ with $m \geq n$ nonzeros can be reordered so the CSR SpMV algorithm has $O(\log n)$ depth and $O(1 + m/B + n/M^{1-\alpha})$ sequential cache complexity.*

Notice that for $B \leq M^{1-\alpha}$ (likely in practice), the m/B term dominates so the number of cache misses is asymptotically optimal (no more than needed to scan the array entries in order).

Proof of Theorem 8.1: We will use a modified CSR representation for all matrices stored in the tree. For symmetric matrices, we only store the lower triangular entries and diagonals for nodes of size $r < \log^{1/(1-\alpha)} n$, and all entries for larger nodes. The idea is that the number of entries in the larger matrices is small enough that we can store them twice or use a pointer to the second copy without significantly affecting space or cache complexity. As mentioned above, we modify CSR so it does not store any information for empty rows.

Consider a matrix at a node of T with size r and with e entries assigned to it. Because the range of columns is bounded by r, all column and row indices can be stored relative to the lower bound of the range using $O(\log r)$ bits. This means the matrix can be stored in ew bits for the values and $O(e \log r)$ bits for the indices. We also need to store the pointers to the children of the node. For this, we assume that the memory for nodes are allocated one after another in a postorder traversal of the tree. This means to point to a child the structure only has to point within the memory used by this subtree. This is certainly bounded by $O(wr^2)$ bits and therefore we can use a $O(\log r)$ bit pointer for each child. We can also use $O(\log r)$ bits to specify the range limits of each child, which we charge to the parent even though stored with the child. Therefore, the total space required by the node with c children is $ew + O((e+c)\log r)$. Now if we organize the tree so the nodes grow doubly exponentially $r_i = 2^{2^i}$, $(2, 4, 16, 256, 65536, \ldots)$, a node at level i captures all edges that were cut in the binary separator tree above size $2^{2^{i-1}}$ and up to 2^{2^i}. Using the separator bounds, and counting per pairwise split, we have for a node at level i, $e_i = \sum_{j=2^{i-1}+1}^{2^i} \eta(j) \times O(2^{\alpha j})$, where $\eta(j) = 2^{2^i - j}$ is the number of splits at the binary tree level j. This sum is bounded by $O(2^{2^{\alpha(i-1)}} 2^{2^{i-1}}) = O(2^{2^{\alpha(i-1)} + 2^{i-1}})$ since the terms of the sum geometrically decrease with increasing j. We also have $c_i = 2^{2^{i-1}}$ for the number of children at level i. There are n/r_i nodes at level i and therefore the total space in bits for pointers is bounded by:

$$\begin{aligned} S(n) &= \sum_{i=0}^{\log\log n} O\left(\frac{n}{r_i}(e_i + c_i)\log r_i\right) \\ &= \sum_{i=0}^{\log\log n} O\left(\frac{n}{2^{2^i}}(2^{(2^{\alpha(i-1)}+2^{i-1})} + 2^{2^{i-1}})2^i\right) \end{aligned}$$

For $\alpha < 1$, this sum geometrically decreases, so for asymptotic analysis, we need only consider $i = 0$ and therefore $S(n) = O(n)$. When we include the space for the matrix values and convert from bits to words, the total space is $m + O(n/w)$. We note that we can store matrices with size $r \geq \log^{1/(1-\alpha)} n$ using two nonzeros per symmetric entry without affecting the asymptotic bounds. This is because there are at most $O(n/\log n)$ nonzeros in matrices of that size so we can use a pointer of size $O(\log n)$ bits to point to the other copy, or if $w = O(\log n)$, we can store the duplicates directly.

We now consider bounds on the sequential cache complexity. The argument is similar to the argument for the CSR format [BCG+08]. We separate the misses into the accesses to the matrix entries and to the input and output vectors x and y. Recall that all tree nodes are stored in post-order with respect to the tree traversal, and at the nodes, the elements within each matrix are stored in CSR format. Since the CSR algorithm visits the matrix in the order it is stored, the algorithm visits all elements in the order they are laid out. When including the $O(n/w)$ words for indices in the structure, which are also visited in order, visiting the matrix causes a total of $m/B + O(n/(Bw))$ misses. For larger nodes in the tree where $r \geq \log^{1/(1-\alpha)} n$, we store duplicate entries, but for the same reason, this is a lower order term in the space and also a lower order term in cache misses. This leaves us to consider the number of misses from accessing x and y. For the sake of analysis, we can partition the leaves into blocks that fit into the cache, where each such block is executed in order by the algorithm. We therefore only have to consider edges that go between blocks. By the same argument as in [BCG+08], the number of such edges (entries) is bounded by $O(n/M^{1-\alpha})$ each potentially causing a miss. The total number of misses is therefore bounded by $m/B + O(1 + n/(Bw) + n/M^{1-\alpha})$.

Finally, we consider the depth of the algorithm. We assume that the SpMV for all nodes of size $r \geq \log^{1/(1-\alpha)} n$ run in parallel since they are stored with both symmetric entries. Such a SpMV runs in $O(\log n)$ depth. For $r < \log^{1/(1-\alpha)} n$, we run the SpMV on the skyline format sequentially. The total time is bounded asymptotically by the size, and all these small multiplies can run in parallel. This is the dominating term giving a total depth of $O(\log^{1/(1-\alpha)} n)$. ■

8.2 Combinatorial Multigrid

To study how the improvements in SpMV performance benefit an actual iterative method, we consider Combinatorial Multigrid (CMG), a recently introduced variant of Algebraic Multigrid (AMG) [KM09, KM08, KMT09] providing strong convergence guarantees for symmetric diagonally dominate linear systems [Kou07, ST04, KM09, KMT09, KMP10]. Our choice is motivated by the potential for immediate impact on the design of industrial strength code for important applications. In contrast to AMG, CMG offers strong convergence guarantees for the class of symmetric diagonally dominant (SDD) matrices [Gre96, BHV04, ACST06], and under certain conditions for the even more general class of symmetric M-matrices [DS08]. The convergence guarantees are based on recent progress in spectral graph theory and combinatorial preconditioning (see for example [BH03], [Kou07]). At the same time, linear systems from these classes play an increasingly important role in a wave of new applications in computer vision [Gra06, TM06, KMT09], and medical imaging in particular [TKI+08]. Multigrid algorithms are commonly used as preconditioners to other iterative methods. The idea of implementing the preconditioner in single precision has been explored before, but the effects on convergence are in general unpredictable [BDK+08]. However, in the case of CMG, switching to single precision has provably no adverse effects. In summary, CMG can benefit from our fastest SpMV primitive, which exploits both symmetry and precision reduction, in applications that are well suited for the diagonal hierarchical blocking approach.

A thorough discussion of multigrid algorithms is out of the scope of this paper. There are many excellent survey papers and monographs on various aspects of the topic and among them [BHM00, TSO00]. The purpose of this section is to discuss aspects of the parallel implementation that are specific to CMG, but at the same time, convince the reader that the performance improvements we see for CMG are expected to carry over to other flavors of multigrid.

8.2.1 CMG Description and Parallel Implementation Details

Similarly to AMG, the CMG algorithm consists of the setup phase which computes a multigrid hierarchy, and the solve phase. The CMG setup phase constructs a hierarchy of SDD matrices $A = A_0, \ldots, A_i$. As with most variants of AMG, CMG uses the *Galerkin condition* to construct the matrix A_{i+1} from A_i. This amounts to the computation of a restriction operator $R_i \in \mathbb{R}^{\dim(A_i)\times\dim(A_{i+1})}$, and the construction of A_{i+1} via the relation $A_{i+1} = R_i^T A_i R_i$. CMG constructs the restriction operator R_i by grouping the variables/nodes of A_i into $\dim(A_{i+1})$ disjoint clusters and letting $R(i,j) = 1$ if node i is in cluster j, and $R(i,j) = 0$ otherwise. This simple approach is known as *aggregate-based* coarsening, and it has recently attracted significant interest due to its simplicity and advantages for parallel implementations [Gra08, MN08]. Classic AMG constructs more complicated restriction operators that can be viewed as (partially) overlapping clusters. The main difference between CMG and other AMG variants is the algorithm for clustering, which in the CMG case is combinatorially rather than algebraically driven. The running time of the CMG setup phase is negligible comparing to the actual MG iteration, so we do not further discuss it in this paper. The reader can find more details in [KMT09].

The solve phase of CMG, which is dominated by SpMV operations, is quite similar to the AMG solve phase; the pseudo-code is given in Figure 8.2.1. When $t_i = 1$, the algorithm is known in the MG literature as the V-cycle, while when $t_i = 2$, it is known as the W-

Algorithm 8.2.1 The CMG Solve Phase

```
function $x_i = \text{CMG}(A_i, b_i)$
 1: $D = \text{diag}(A)$
 2: $r_i = b_i - A_i(D^{-1}b)$
 3: $b_{i+1} = Rr_i$
 4: $z = \text{CMG}(A_{i+1}, b_{i+1})$
 5: for $i = 1$ to $t_i - 1$ do
 6:     $r_{i+1} = b_{i+1} - A_{i+1}z$
 7:     $z = z + \text{CMG}(A_{i+1}, r_{i+1})$
 8: end for
 9: $x = R^T z$
10: $x = r_i - D^{-1}(A_i x - b)$
```

cycle. It has been known that the aggregate-based AMG does not exhibit good convergence for the V-cycle. The theory in [Kou07] essentially proves that more complicated cycles are expected to converge fast, without blowing up the total work performed by the algorithm. This is validated by our experiments with CMG where we pick

$$t_i = \max\left\{\left\lceil \frac{nnz(A_i)}{nnz(A_{i+1})} - 1\right\rceil, 1\right\}.$$

Here $nnz(A)$ denotes the number of nonzero entries of A. This choice for the number of recursive calls, combined with the fast geometric decrease of the matrix sizes, targets a geometric decrease in the total work per level.

In our parallel implementation, we optimized the CMG solve phase by using different SpMV implementations for different matrix sizes. When the matrix size is larger than 1K, we use the blocked version of SpMV, and when it is smaller than that, we resort to the plain parallel implementation, where the matrix is stored in full and we compute each row in parallel. The reason is that the blocked version of SpMV has higher overhead than the simple implementation for smaller matrices.

In our experiments, we found that a choice of $t'_i = t_i + 1$ improves (in some examples) the sequential running time required for convergence by as much as 5%. However, it redistributes work to lower levels of the hierarchy where, as noted above, the SpMV speedups are smaller. As a result, the overall performance gains for CMG are less significant with this choice.

8.2.2 Single vs. Double Precision CMG

The CMG solve phase is the implicit inverse of a symmetric positive operator B. The condition number $\kappa(A, B)$ can therefore be defined, and it is well-understood that it characterizes the rate of convergence of the preconditioned CG iteration [Axe94].

Recall that the CMG core works with the assumption that the system matrix A is SDD. We form a single precision matrix $\hat{A}$ from the double precision matrix A as follows; we decompose A into $A = D + L$, where L has zero (in double precision) row sums and D is a diagonal matrix with non-negative entries. We form $\tilde{D}$ by casting the positive entries of D into single precision. We form $\tilde{L}$ by casting the off-diagonal entries of L into single precision, adding them in the order they appear using single precision, and then negating the sum and setting it to the corresponding diagonal entry of $\tilde{L}$. Finally, we let $\tilde{A} = \tilde{D} + \tilde{L}$. This construction guarantees that $\tilde{A}$ is numerically diagonally dominant and thus positive definite.

Substituting a double-precision hierarchy $A_0, \ldots, A_d$ by its single-precision counterpart $\tilde{A}_0, \ldots, \tilde{A}_d$ in effect changes the symmetric operator B to a new operator $\hat{B}$, which is also symmetric. By an inductive (on the number of levels) argument, it can be shown that

$$\kappa(B, \tilde{B}) \leq \max_i \kappa(A_i, \tilde{A}_i).$$

Using the Splitting Lemma for condition numbers [BH03], it is easy to show that

$$\kappa(A, \tilde{A}) \leq \left(\max_i \left\{ \frac{D_{i,i}}{\tilde{D}_{i,i}}, \frac{\tilde{D}_{i,i}}{D_{i,i}}, \max_{j \neq i} \left\{ \frac{|L_{i,j}|}{|\tilde{L}_{i,j}|}, \frac{|\tilde{L}_{i,j}|}{|L_{i,j}|} \right\} \right\} \right)^2 .$$

Under reasonable assumptions for the range of numbers used in A, we get $\kappa(B, \tilde{B}) < 1 + 10^{-7}$. Using the transitivity of condition numbers, we get

$$\kappa(A, \tilde{B}) \leq \kappa(A, B)\kappa(B, \tilde{B}) \leq \kappa(A, B)(1 + 10^{-7}).$$

It is known that the condition number of a pair (A, B) is the ratio of the largest to the smallest generalized eigenvalue of (A, B). The above inequality can in fact be extended to show that *each* generalized eigenvalue of the pair (A, B) is within a $(1 + 10^{-7})$ factor of the corresponding generalized eigenvalue of $(A, \tilde{B})$. Thus, the preconditioned CG is expected to have an almost identical convergence, independent of whether B or $\tilde{B}$ is the preconditioner. the two preconditioned CG iterations are virtually indistinguishable with respect to their convergence rates.

8.3 Implementation and Evaluation

This section describes an implementation of an SpMV based on hierarchical diagonal blocking and a study of its performance compared to other related variants.

8.3.1 Implementation of SpMV

We implemented SpMV routines for symmetric matrices using the descriptions from Section 8.1. The implementation stores a matrix as groups of on-diagonal entries, diagonal-block entries, and off-block entries (similar to Figure 8.2 with only 2 inner-node levels and a level of leaf nodes). The diagonal blocks in the first level are $\sim$ 32K in size (to take advantage of caching) and the leaves correspond the singletons along the matrix's diagonal. This representation allows for a simple implementation which delivers good performance in practice.

The two main ideas from previous sections are precision reduction and diagonal blocking. To understand the benefits of these ideas individually, we study the following variants: the sequential program using double-precision numbers "seq. (double)" is our baseline implementation (more details below). The simple parallel program for double-precision numbers "simple par. (double)" computes the rows in parallel. The corresponding version for single-precision numbers is known as "simple par. (single)." We have two variants of the hierarchical diagonal blocking routines, one for double-precision numbers "blocked par. (double)" and one for single-precision numbers "blocked par. (single)". *The names inside quotation marks are abbreviated names used in all the figures.*

The baseline implementation is a simple sequential program similar to what is shown in Figure 8.1. We optimized the code slightly by applying one level of loop-unrolling to the inner loop. Note that although the code is simple, its performance matches, within 1%, that of highly optimized kernels for SpMV, such as Intel Math Kernel Library [int10b]. We decided to work with our own implementation because of the flexibility in changing and instrumenting the code (e.g., for collecting statistics).

All versions of our parallel programs were written in Cilk++, a language similar to C++ with keywords that allow users to specify what should be run in parallel [int10a]. Cilk++'s runtime system relies on a work-stealing scheduler, a dynamic scheduler that allows tasks to be rescheduled dynamically at low overhead cost. Our benchmark programs were compiled with Intel Cilk++ build 8503 using the optimization flag `-O2`. To avoid overhead in the Cilk++'s runtime system, we compiled the baseline sequential programs with GNU `g++`

version 4.4.1 using the optimization flag `-O2`.[1]

8.3.2 Experimental Setup

Testbed. We are interested in understanding the performance characteristics of SpMV and CMG solvers on three recent machine architectures: the Nehalem-based Xeon, the Intel Harpertown, and the AMD Opteron Shanghai. A brief summary of our test machines is presented in Table 8.1. Our measurements were taken with hyperthreading turned off. Even though hyperthreading gives a slight boost in performance (though less than 5%), the timing numbers were much more reliable with it turned off.

Machine Model	Speed	Layout	Agg. Bandwidth	
	(Ghz)	(#chips$\times$#cores)	1 core	8 cores
Intel Nehalem X5550	2.66	2×4	10.5	27.9
Intel Harpertown E5440	2.83	2×4	2.8	6.4
AMD Shanghai 2384	2.70	2×4	4.9	10.7

Table 8.1: Characteristics of the architectures used in our study, where clock speeds are reported in Ghz and 1- and 8-core aggregate bandwidth numbers in GBytes/sec. For the aggregate bandwidth, we report the performance of the *triad* test in the University of Virginia's STREAM benchmark [McC07], compiled with `gcc -O2` and using `gcc`'s OpenMP.

Among these architectures, the Intel Nehalem is the current flagship, which shows significant improvements in bandwidth over prior architectures. For this reason, this work focuses on our performance on the Nehalem machine; we include results for other architectures for comparisons as our techniques benefit other architectures as well.

Datasets. Our study involves a diverse collection of large sparse matrices, gathered from the University of Florida Matrix Collection [Dav94] and a collection of mesh matrices generated by applications in vision and medical imaging. We present a summary of these matrices in Table 8.2. These matrices are chosen so that for the majority of them, neither the vectors nor the whole matrix can fit entirely in cache; smaller matrices are also included for comparison.

For the CMG experiments, since the CMG solver requires the input matrix to be SDD, we replace each off-diagonal entry with a negative number of the same magnitude, and we adjust

[1]We have also experimented with the Intel compiler and found similar results.

Matrix	#rows/cols	#nonzero
2d-A	999,999	4,995,995
3d-A	999,999	6,939,993
af_shell10	1,508,065	52,672,325
audikw_1	943,695	77,651,847
bone010	986,703	71,666,325
ecology2	999,999	4,995,991
nd24k	72,000	28,715,634
nlpkkt120	3,542,400	96,845,792
pwtk	217,918	11,634,424

Table 8.2: Summary of matrices used in the experiments.

the diagonals to get zero row-sums. The perturbation does not affect the SpMV performance, as the matrix structure remains unchanged, but it allows us to study the performance of CMG on various sparse patterns.

All matrices in the study are ordered in the best possible ordering we are able to find. Each matrix is reordered using a number of heuristics and we keep the ordering that yields the best baseline performance. For each matrix, we use the same ordering when comparing SpMV schemes. We discuss the effects of separator quality in Section 8.3.5.

8.3.3 Performance of SpMV

The first set of experiments concerns the performance of SpMV. In these experiments, we are especially interested in understanding how the ideas outlined in previous sections perform on a variety of sparse matrices.

Throughput. Figure 8.3 and Table 8.3 show the performance (in GFlops) and the speedup achieved by various SpMV routines on the matrices in our collection. Several things are clear. First, on all these matrices, a simple parallel algorithm speeds up SpMV by 3.4x–4.5x. In fact, without any data reduction, we cannot hope to improve the performance much further, because as will be apparent in the next discussion, the simple parallel algorithm operates near the peak bandwidth.

Second, but more importantly, both hierarchical diagonal blocking and precision reduction can help enhance the speed of SpMV, but *neither idea alone yields as much performance improvement as their combination.* By replacing double-precision numbers with single-

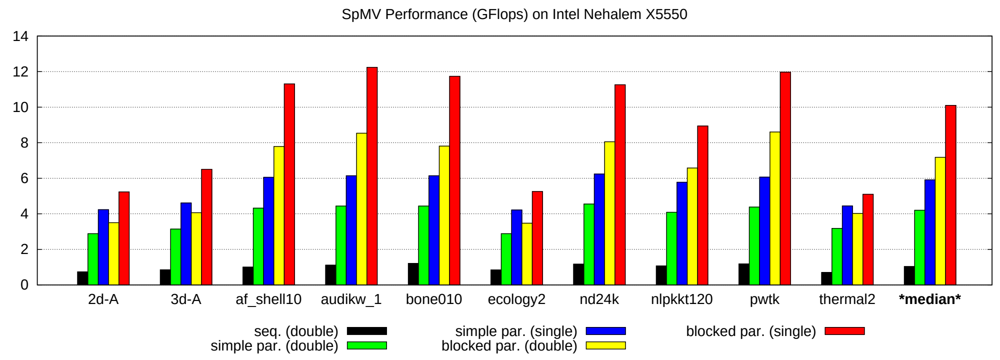


Figure 8.3: Performance of different SpMV routines (**in GFlops**) on a variety of matrices.

Matrix	**Speedup** simple par. (double)	**Speedup** blocked par. (single)
2d-A	3.9x	7.1x
3d-A	3.7x	7.6x
af_shell10	4.3x	11.3x
audikw_1	4.0x	11.0x
bone010	3.7x	9.7x
ecology2	3.4x	6.2x
nd24k	3.9x	9.6x
nlpkkt120	3.8x	8.4x
pwtk	3.7x	10.1x
thermal2	4.5x	7.3x

Table 8.3: Speedup numbers of parallel SpMV on an 8-core Nehalem machine as compared to the sequential baseline code.

precision numbers, we use 4 bytes per matrix entry instead of 8. Furthermore, by using the hierarchical diagonal blocking with the top-level block size $\sim$ 32K, we can represent the indices of the entries in the diagonal blocks using 16-bit words, a saving from 32-bit words used to represent matrix indices in a normal CSR format. Diagonal blocking can also take advantage of symmetry: each digonal blocks can be stored in the skyline format, which halves the number of entries (both indicies and values) we have to store. Combining these ideas, we not only further reduce the bandwidth but also improve the cache locality due to blocking. Shown in Table 8.4 is the memory footprint of the different representations. By applying the blocking on these matrices, the footprint can be reduced by more than 1.5x and can be further reduced by precision reduction. This is reflected in the additional speedup of more than 2x in the speedup of the single-precision blocked parallel version over the speedup of the simple double-precision parallel code.

Matrix	**Memory Access (MBytes)**		
	CSR/double	blocked/double	blocked/single
2d-A	80	56	36
3d-A	103	67	43
af_shell10	657	313	193
audikw_1	951	426	261
bone010	880	404	251
ecology2	80	56	36
nd24k	346	164	106
nlpkkt120	1212	589	367
pwtk	143	65	40
thermal2	128	85	55

Table 8.4: Total memory accesses (**in MBytes**) to perform one SpMV operation using different representations.

Scalability. Presented in Figures 8.4 and 8.5 are speedup and bandwidth numbers for different SpMV routines. The speedup on i cores is how much faster a program is on i cores than on 1 core running the same program. First and most importantly, *blocked parallel single precision scales the best on all three machines.* On the Nehalem, it achieves a factor of almost 7x compared to approximately 4x for the simple double-precision parallel SpMV. Furthermore, the trend is similar between Nehalem and Shanghai, which both have more memory channels and higher bandwidth than the Harpertown. On the Harpertown, all the benchmarks saturate at 4 cores, potentially due to the limited bandwidth.

Second, reducing the memory footprint (hence the bandwidth requirement) is key to improving the scalability. As Figure 8.5 shows, the simple parallel SpMV seems to be compute bound on 1 core but runs near peak bandwidth on 8 cores, suggesting that further performance improvement is unlikely without reducing the bandwidth requirements. But, as noted earlier, the blocked schemes have substantially smaller memory footprint than the simple scheme. For this reason, the blocked schemes are able to achieve better FLOPS counts and scalability even though they do not operate near the peak bandwidth.

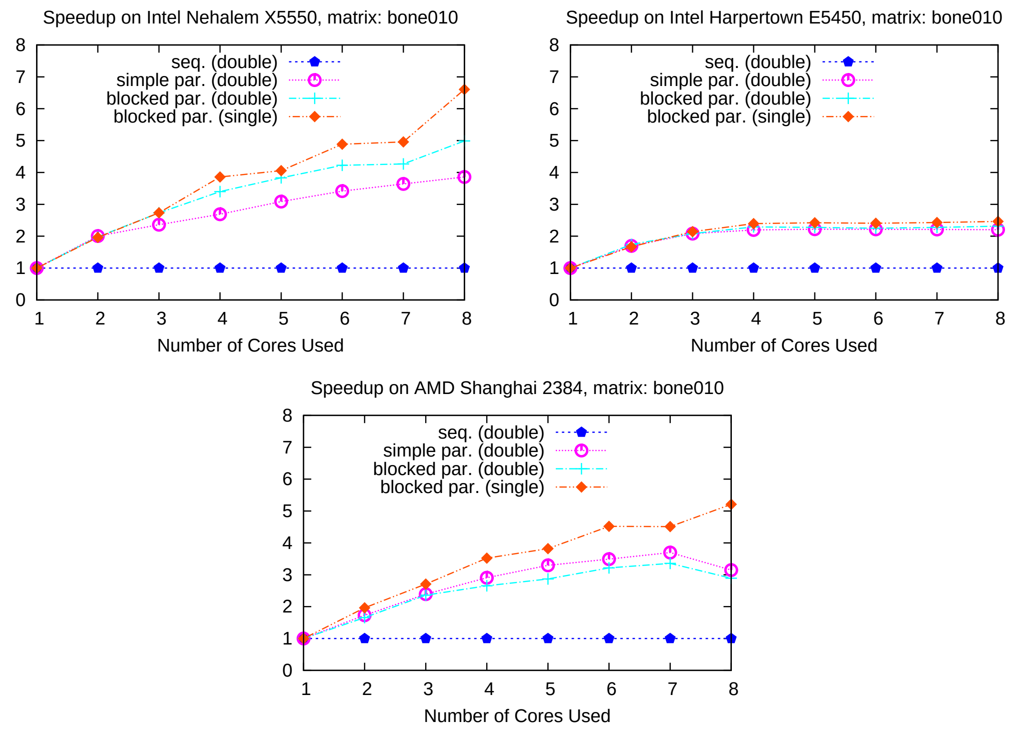


Figure 8.4: Speedup factors of SpMV on Intel Nehalem X5550, AMD Shanghai 2384, and Intel Harpertown E5440 as the number of cores used is varied.

8.3.4 Performance and Convergence of CMG

Figure 8.6 shows the performance of *one call* to three CMG programs, differing in the SpMV kernel used. The precision of scalars and vectors used by CMG match that of its SpMV kernel. In the parallel implementations, vector-vector operations in the CMG programs are also parallelized, when possible, in a straightforward manner.

From the figure, two things are clear. First, the speedup—the ratio between the performance of the baseline sequential program and the parallel one—varies with the linear system being solved; however, on all datasets we consider here, the speedup is more than 3x, with the best case reaching beyond 6x. Second, the speedup of the CMG solver seems to be proportional to the speedup of SpMV, but not as good. This finding is consistent with the fact that the largest fraction of the work is spent in SpMV, while part of the work is spent on operations with more modest speedups (e.g., vector-vector operators and SpMV operations on smaller matrices).

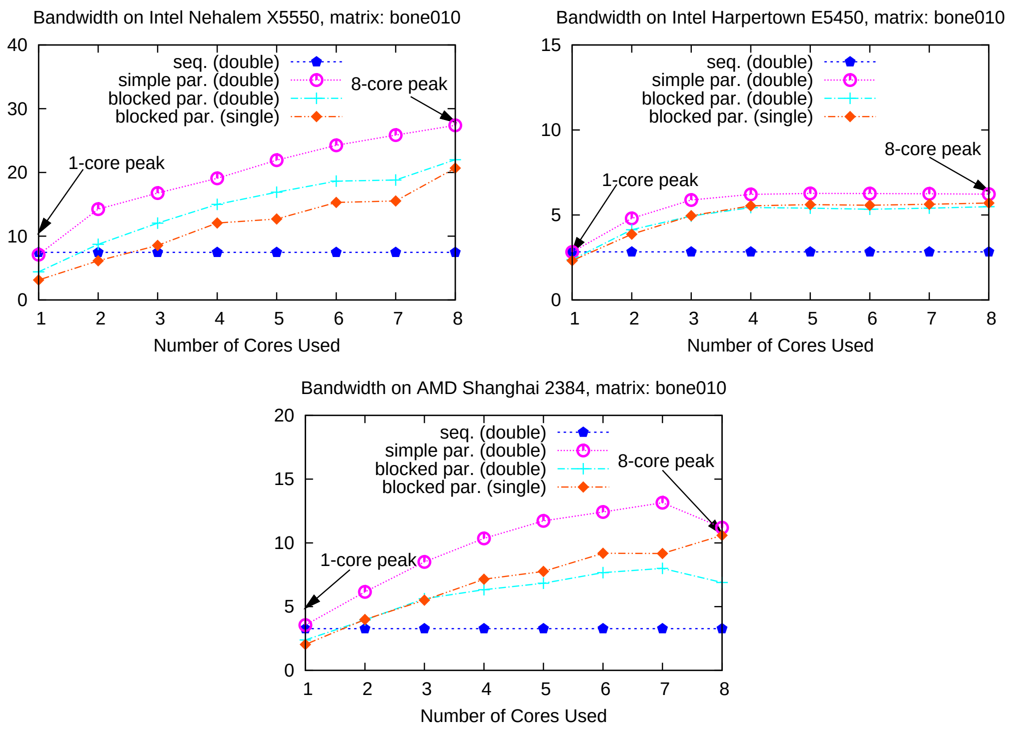


Figure 8.5: Bandwidth consumption of SpMV (**in GBytes/sec**) on Intel Nehalem X5550, Intel Harpertown E5440, and AMD Shanghai 2384 as the number of cores used is varied. The peak 1- and 8-core bandwidth numbers are from Table 8.1.

The CMG is used as a preconditioner in a Preconditioned Conjugate Gradients (PCG) iteration. In Table 8.5, we report the number of PCG iterations required to compute a solution x such that the relative residual error satisfies $\|Ax - b\|/\|b\| < 10^{-8}$, for various matrices and three different b-sides. The first column corresponds to a random vector b, the second to Ab and the third to an approximate solution of $Ax = b$, for the same random b. We note that the reported convergence rates are preliminary. Improvements may be possible as long as the hierarchy construction abides by the sufficient and necessary conditions reported in [KM08]. One call to CMG is on average 5–6 times slower than one call to SpMV. Most of the matrices have a particularly bad condition number and standard CG without preconditioning would require thousands of iterations to achieve the same residual error.

As predicted by the theory in Section 8.2.2, CG preconditioned with double-precision CMG is virtually indistinguishable from CG preconditioned with single-precision CMG; the number of iterations for convergence differs by at most 1 in all our experiments. We have also found

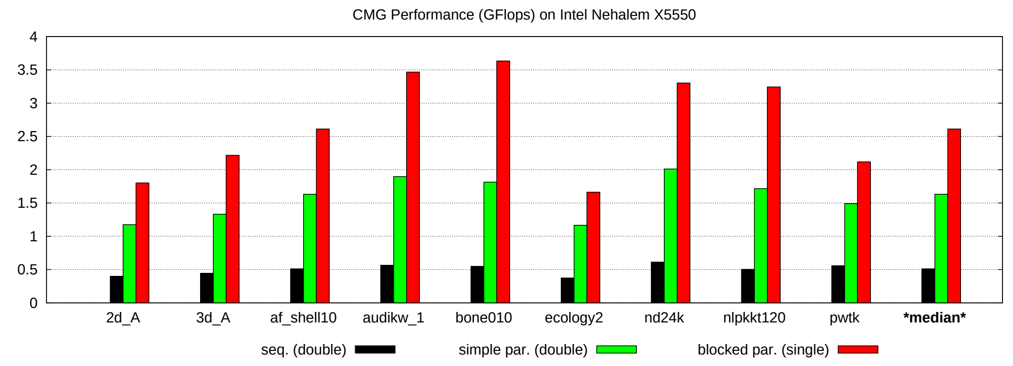


Figure 8.6: Performance of a CMG solve iteration (**in GFlops**) on different linear systems.

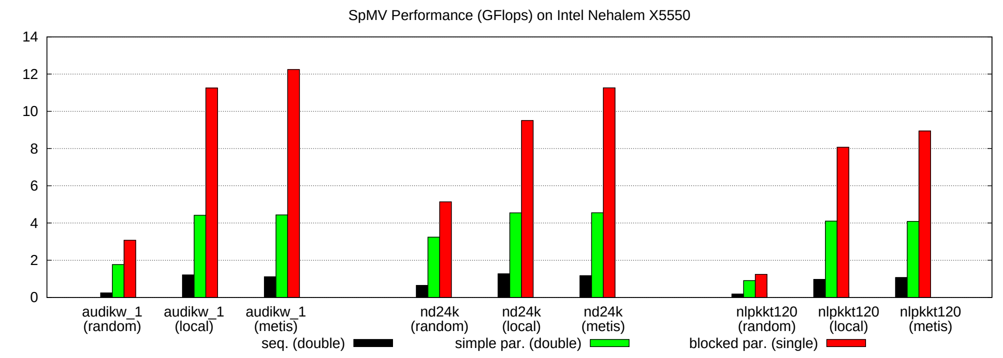


Figure 8.7: Performance of SpMV routines (**in GFlops**) with different ordering heuristics.

that further improvements can be found by using a single-precision implementation of CG to drive the error down to 10^{-6} and then switching to the double-precision mode. In Table 8.5, we report the running times of *one call* to PCG, with CG implemented in single precision and double precision—the preconditioner CMG is implemented consistently in single precision.

8.3.5 Effects of Separator Quality

Our results thus far rely on the assumption that the input matrices are given in a good separator-ordering. Often, however, the matrices have good separators but are not prearranged in such an ordering. In this section, we explore various heuristics for computing a good separator-ordering and compare their relative performance with respect to SpMV.

We begin by defining two abstract measures of the quality of an ordering. The first measure,

Matrix	**#iterations**			**PCG run time per call**	
	random b	Ab	A^+b	P-single-CG	P-double-CG
2d-A	42	34	48	24.15	31.1
3d-A	37	32	37	24.3	31.5
af_shell10	26	23	30	195.3	231.3
audikw_1	19	15	17	205.0	245.8
ecology2	49	37	55	25.5	32.2
nlpkkt120	26	20	28	203.2	256.8

Table 8.5: PCG: number of iterations required for convergence of error to 10^{-8} and running time per call **in milliseconds**.

called the ℓ*-distance*, is inspired by previous work on graph compression using separator trees [BBK03]. The ℓ-distance is an information-theoretic lower bound on the average number of bits needed to represent the index of an entry. This measure therefore indicates how well the ordering compresses. Formally, for a matrix M,

$$\ell(M) := \frac{1}{\#nnz} \sum_{(i,j)\in M} \log_2 |i - j + 1|.$$

Simpler than the first, the second measure—denoted by "off"—is simply the percentage of the nonzero entries that fall off the first-level blocks. This measure tells us what fraction of the nonzero elements has to resort to the simple parallel scheme and cannot benefit from the blocks.

As we already discussed in Section 8.1, at the heart of a separator ordering is a separator tree—a fully balanced tree of edge-separators for the graph of the matrix. For the study, we consider the following graph-partitioning and reordering heuristics: (1) "local," a bottom-up contraction heuristic (known in the original paper as bu) [BBK03]; (2) METIS, an algorithm which recursively applies the METIS partitioning algorithm [KK98]; and (3) a random ordering of the vertices.

Table 8.6 shows statistics for these heuristics on three of the matrices used in previous sections. On both the ℓ-distance and off-block measures, it is clear that METIS produces superior orderings than the local heuristic does on all of the matrices considered; however, the local heuristic is significantly faster than METIS, both running sequentially—and as we will see next, both schemes yield comparable SpMV performance. In terms of parallelization potential, we were unable to run parMETIS on our Nehalem machine. Yet, the local heuristic shows good speedup running on 8 cores, finishing in under 3 seconds on the largest matrix with almost 100 million entries and exhibiting more than 5x speedup over 1 core.

Matrix	nnz/row	Random		Local				METIS		
	(avg.)	ℓ	off	ℓ	off	T_1	T_8	ℓ	off	T_1
audikw_1	82.3	17.5	92.6%	7.6	9.0%	11.1	1.9	6.8	3.6%	76.1
nd24k	399.0	13.9	93.5%	9.5	36.1%	5.0	0.8	8.5	21.4%	12.0
nlpkkt120	26.9	19.2	96.6%	7.5	11.9%	15.3	2.6	6.3	5.3%	230.5

Table 8.6: Statistics about different ordering heuristics: ℓ is the ℓ-distance defined in Section 8.3.5 and off is the percentage of the entries that fall off the diagonal blocks. The timing numbers (**in seconds**, T_1 for the sequential code and T_8 for the parallel code on 8 cores) on the Nehalem are reported.

We show in Figure 8.7 how the different ordering heuristics compare in terms of SpMV performance. First but unsurprisingly, the random ordering, which we expect to have almost no locality, performs the worst on all three SpMV algorithms. Second, as can be seen from the stark difference between the random ordering and the other two schemes, a good separator-ordering benefits *all* algorithms, not just the HBD scheme. Third but most importantly, the SpMV algorithms are "robust" against small differences in the separator's quality: on all algorithms, there is no significant performance loss when switching from METIS to a slightly worse, but faster to compute, ordering produced by the local heuristic.

8.4 Conclusions

This chapter described a sparse matrix representation which in conjunction with precision reduction, forms the basis for high-performance SpMV kernels. We evaluated their performance both as stand-alone kernels and on CMG, showing substantial speedsup on a diverse collection of matrices.

Chapter 9

Parallel and I/O Efficient Set Cover, Max Cover, and Related Problems

In this chapter, we build on the algorithms from Chapter 5 to give approximation algorithms for set cover and related problems that are both parallel and I/O efficient.

The past few years have witnessed the proliferation of parallel machines, with tens of cores readily available in commodity machines; a similar time frame also saw a dramatic increase in disk bandwidths through the advent of solid-state drives (SSDs). These two seemingly disparate trends might seem to lead to very different directions in algorithms design, but fortunately, often approaches that are good for parallelism are also good for external memory [CGG+95]. Roughly speaking, this is because techniques that generate parallelism can enable algorithms to hide latency and batch-access external blocks of memory.

From a practical perspective, set cover and its variants have many applications that can require very large data [CKW10], including data analysis, information retrieval, fault testing, and allocating wavelength in wireless communication. It thus seems well-suited for parallel and I/O efficient algorithms. From a theoretical perspective, although set cover is one of the most fundamental and well-studied problems in optimization and approximation algorithms, we know of no I/O efficient solutions for the general case (i.e. when neither the sets nor the elements fit in memory). Recent work of Cormode, Karloff, and Wirth (CKW) has developed an efficient algorithm for the case when the elements—but not the sets—fit in memory [CKW10]. We present a solution that is both I/O-efficient and parallel.

Our (randomized) algorithm for (weighted) set cover extends recent results for parallel set cover [BPT11] (BPT) to I/O models. As with other previous parallel algorithms for the problem [BRS94, RV98], and also as with the CKW approach, the idea is based on bucketing costs in powers of $(1 + \varepsilon)$. The BPT approach, however, uses maximal nearly-independent sets to achieve good parallelism and hence I/O efficiency within a step while being *work-efficient*—each edge is processed on average a constant number of times.[1] In the I/O models where permutations are as expensive as sorting, the algorithm achieves the same bound as sorting. Our algorithm gives an $(1 + \varepsilon)(1 + \ln n)$-approximation for arbitrary $\varepsilon > 0$ and hence is essentially optimal. In addition to set cover, as shown in [BPT11], the same sequence of sets can be used as a solution to max cover and min-sum set cover. For max cover, this sequence is prefix optimal: for any prefix of length k, this prefix is a $(1 - \frac{1}{e} - \varepsilon)$-approximation to the max k-cover problem.

We analyze the cost in the parallel cache oblivious (PCO) model [BFGS11]. The model captures both parallelism and I/O (cache) efficiency by analyzing algorithms in terms of their depth (aka. critical path, span) and cache complexity. The model supports nested fork-join parallelism starting with a single sequential strand and allowing arbitrary dynamic nesting of parallel loops or forks and joins. Upper bounds on cache complexity in the PCO model imply the same cache complexity in the sequential cache oblivious model [FLPR99], and hence the external memory (EM) model when $M \geq B^2$ (the standard tall cache assumption) [Vit01].

In the PCO model (and hence CO and EM models), the cache (I/O) complexity of our algorithm is $O(\frac{W}{B} \log_{M/B} \frac{W}{B})$, where W is the size of the input (number of edges). This matches the sorting lower bounds for all models. Furthermore, it has polylogarithmic parallel depth. The PCO bounds also imply equivalent bounds on total misses on parallel machines with various cache hierarchies including shared caches, distributed caches and trees of caches [BFGS11].

We have implemented and experimented with two variants of our algorithm, one for shared-memory parallel machines and one for external memory. The only difference between the two variants is how we implement the communication steps. We experiment with the parallel variant on a modern 48-core multicore with just enough memory to fit all our instances (64 Gbytes), and with the external memory variant, on a more modest machine using an external solid-state drive. We test the algorithms with several large instances with up to 5.5 billion edges. For the parallel version, we are able to achieve significant speedup over a fast sequential implementation. In particular, we compare to the CKW sequential algorithm

[1]Here, edges refer to the bipartite graph representation of a set-cover instance with sets on one side, ground elements on the other, and an edge when a set includes an element.

which is already significantly faster than the greedy algorithm. For our largest graph with about 5.5 billion edges, the algorithm runs in under a minute. In fact, it runs faster than optimized parallel sorting code [SSP07] on the same sized data, and over 20x faster than sequential sorting code. For the max k-cover problem, we empirically show that it is often possible to speedup the computation by more than a factor of 2 by stopping the algorithm early when k is known and is small relative to the set cover's solution size. For the sequential I/O variant, we are able to achieve orders of magnitude speedups over the results of CKW when neither the sets nor elements fit in memory. When the elements fit in memory, the CKW algorithm is faster. With regards to quality of the results (number of sets returned), our algorithm returns about the same number of sets as the other algorithms.

Parallel Primitives

We need primitives such as sorting, prefix computation, merge, filter, map for the set cover algorithm. Parallel algorithms with optimal cache complexity in the PCO model and polylogarithmic depth can be constructed for these problems (for construction, see [BFGS11]). The cache complexity of sorting on an input instance n in the PCO model is $\text{sort}(n; M, B) = O(\frac{n}{B}\log_{M/B}\frac{n}{B})$, while the complexity of the other primitives is $O(n/B)$. We use $\text{sort}(n)$ as shorthand. All primitives have $O(\log^2 n)$ depth.

9.1 Algorithm Design and Implementation

This section describes an efficient implementation of the BPT set cover algorithm [BPT11] in the PCO model, implying good I/O complexity in other related models. We begin by presenting an algorithmic description, which satisfies Theorem 9.1. Following that, we discuss implementation details in optimizing this algorithm for different hardware setups.

Throughout this section, let $W = \sum_{S\in\mathcal{S}} |S|$ be the sum of the set sizes.

Theorem 9.1 (Parallel and I/O Efficient Set Cover) *The I/O (cache) complexity of the approximate set cover algorithm on an instance of size W is $O(\text{sort}(W))$ and the depth is polylogarithmic in W. Furthermore, this implies an algorithm for prefix-optimal max cover and min-sum set cover in the same complexity bounds.*

At the core of the BPT algorithm (reproduced in Algorithm 9.1.1 for reference) are the following 3 ingredients—prebucketing, MaNIS, and bucket management—which we discuss in turn. The universe of elements $\mathcal{U}$ is represented as a bitmap indexed by an element identifier.

Algorithm 9.1.1 `SetCover` — Blelloch et al. parallel greedy set cover.

Input: a set cover instance $(\mathcal{U}, \mathcal{F}, c)$ and a parameter $\varepsilon > 0$.
Output: a *ordered* collection of sets covering the ground elements.

```
i. Let $\gamma = \max_{e\in\mathcal{U}} \min_{S\in\mathcal{F}} c(S)$, $W = \sum_{S\in\mathcal{F}} |S|$, $T = \log_{1/(1-\varepsilon)}(W^3/\varepsilon)$, and $\beta = \frac{W^2}{\varepsilon\cdot\gamma}$.
ii. Let $(\mathcal{A}; A_0, \ldots, A_T) = \texttt{Prebucket}(\mathcal{U}, \mathcal{F}, c)$ and $\mathcal{U}_0 = \mathcal{U} \setminus (\cup_{S\in\mathcal{A}} S)$.
iii. For $t = 0, \ldots, T$, perform the following steps:
    1. Remove deleted elements from sets in this bucket: $A'_t = \{S \cap \mathcal{U}_t : S \in A_t\}$
    2. Only keep sets that still belong in this bucket: $A''_t = \{S \in A'_t : |S|/c(S) > \beta \cdot (1-\varepsilon)^{t+1}\}$.
    3. Select a maximal nearly independent set from the bucket: $J_t = \texttt{MaNIS}_{(\varepsilon,3\varepsilon)}(A''_t, \{|a| : a \in A\})$.

    4. Remove elements covered by $J_t$: $\mathcal{U}_{t+1} = \mathcal{U}_t \setminus X_t$ where $X_t = \cup_{S\in J_t} S$
    5. Move remaining sets to the next bucket: $A_{t+1} = A_{t+1} \cup (A'_t \setminus J_t)$
iv. Finally, return $\mathcal{A} \cup J_0 \cup \cdots \cup J_T$.
```

Since we only need one bit of information per element to indicate whether it is covered or not, this can be stored in $O(\frac{|\mathcal{U}|}{\log W})$ words.

Prebucketing. This component (Step ii.) ensures that the ratio between the costliest set and cheapest set is polynomially bounded so that the total number of buckets is kept logarithmic. As described originally (cf. Lemma 4.2 of [BPT11]), this part involves discarding sets that cost more than a threshold, including in the solution all sets cheaper than a certain threshold—and marking the covered elements in the bitmap—then assigning the remaining sets to $O(\log W)$ different buckets by their cost. These operations can be done using filter, sort, and merge, all in less than sort(W) I/O complexity and $O(\log^2 n)$ depth in the PCO model.

MaNIS. Invoked in Step iii of the set cover algorithm, MaNIS finds a subcollection of the sets in a bucket that are almost non-overlapping with the goal of closely mimicking the greedy behavior. Shown in Algorithm 9.1.2 is a modified version of MaNIS algorithm from [BPT11] with the steps annotated with primitives in the PCO model used to implement them. By inspection, it is clear that each round of MaNIS require at most $O(\text{sort}(|G|)$ I/O complexity and depth in the PCO model. As analyzed in [BPT11], for a bucket with W_t edges to start with, MaNIS runs for at most $O(\log W_t)$ rounds—and after each round, the number of edges drops by a constant factor; therefore, we have the following bounds:

Lemma 9.2 *The cache (I/O) complexity in the PCO model of running MaNIS on a bucket with W_t edges is $O(sort(W_t))$, and the depth is $O(sort(W_t)\log W_t)$.*

Bucket Management. The remaining steps in Algorithm 9.1.1 are devoted to bucket man-

Algorithm 9.1.2 $\texttt{MaNIS}_{(\varepsilon,3\varepsilon)}(G, D)$

Input: A bipartite graph $G = (A, N_G(a))$, and a degree function $D(a)$.
A is a sequence of left vertices (the sets), and $N_G(a), a \in A$ are the neighbors of each left vertex on the right.
These are represented as contiguous arrays. The right vertices are represented implicitly as $B = N_G(A)$.
Output: $J \subseteq A$ of chosen sets.

```
1. If $A$ is empty, return the empty set.
2. For $a \in A$, randomly pick $x_a \in_R \{0, \ldots, W_G^7 - 1\}$.   // ensure no collision with high probability
3. For $b \in B$, let $\varphi$ be $b$'s neighbor with maximum $x_a$              // sort and prefix sum
4. Pick vertices of $A$ "chosen" by sufficiently many in $B$:          // sort, prefix sum, and filter

       $J = \{a \in A | \#\{b : \varphi(b) = a\} \geq (1 - 4\varepsilon)D(a)\}.$

5. Update the graph by removing $J$ and its neighbors, and elements of $A$ with too few remaining
   neighbors:
   (1) $\overline{B} = N_G(J)$ (elements to remove)                                  // sort
   (2) $N'_G = \{\{b \in N_G(a) | b \notin \overline{B}\} : a \in A \setminus J\}$                    // sort and filter
   (3) $A' = \{a \in A \setminus J : |N'_G(a)| \geq (1 - \varepsilon)D(a)\}$                          // filter
   (4) $N''_G = \{\{b \in N'_G(a)\} : a \in A'\}$                                     // filter
6. Recurse on reduced graph: $J_R = \texttt{MaNIS}_{(\varepsilon,3\varepsilon)}((A', N''_G), D)$
7. return $J \cup J_R$
```

agement, ensuring the contents of the least-costly bucket is "fresh." We assume the A_t, A'_t and A''_t are stored in the same format as the input for MaNIS (see 9.1.1). Step iii.1 can be accomplished using a sort to order the edges by element index, a merge (with a vector of length $O(|\mathcal{U}|/\log W)$) to match them with the elements bitmap, a filter to remove deleted edges, and another sort to get them back ordered by set. Step iii.2 is simply a filter. The append operation in Step iii.5 is no more expensive than a scan. In the PCO models, these primitives have I/O complexity at most $O(\text{sort}(W_t) + \text{scan}(|\mathcal{U}|/\log W))$ for a bucket with W_t edges. They all have $O(\text{sort}(W))$ depth.

To show the final cache (I/O) complexity bounds, we make use of the following claim:

Claim 9.3 ([BPT11]) *Let W_t be the number of edges in bucket t at the beginning of the iteration which processes this bucket. Then, $\sum W_t = O(W)$.*

Therefore, we have $O(\text{sort}(W))$ from prebucketing, $O(\text{sort}(W))$ from MaNIS combined, and $O(\text{sort}(W) + \text{scan}(\mathcal{U}))$ from bucket management combined. This simplifies to an I/O (cache) complexity of $Q^*(W; M, B) = O(\text{sort}(W; M, B))$ since $U \leq W$. The depth is $O(\log^4 W)$.

9.1.1 Implementation Details

Here we discuss certain design decisions that we made for the two versions of the MaNIS-based set cover algorithm: one optimized for the multicore architecture and the other for the external-memory setting. In both cases we store the adjacency arrays (integers for the elements belonging to each set) contiguously within each array and then across arrays. We maintain a pointer from each set to the beginning of its array.

Parallel Implementation

This implementation targets modern machines with many cores and sufficient RAM to fit and process the dataset if sufficient care is taken to manage the memory. The goal of this implementation is therefore to take advantage of available parallelism and locality, and strike a balance between the computation cost and the memory-access cost. We apply bucket sort and standard prefix computations to put the input sets into the buckets they belong. To implement MaNIS, we make the following observation: a round of MaNIS can be seen as a small number of sorts, as presented, or as handling concurrent priority writes. Our preliminary experiments show that simulating priority writes using compare and swap (CAS) on these sets in a standard way does not produce high contention—and is in fact faster than running sort a few times. This is the tradeoff between computation cost and memory-accesses that we alluded to earlier. We also made efforts to minimize the number of passes over the bitmap and the data.

Disk-optimized Implementation

At the other end of the spectrum, we target a single-core machine with so little fast memory that not even the bitmap—the bit indicator array for the elements—cannot fit in main memory, but this machine has relatively fast disk (e.g., a solid-state drive). To perform external-memory (out-of-core) computations, we resort to STXXL, a C++ reimplementation of the Standard Template Library (STL) for external memory usage [DKS08, SSP07]. STXXL hides from the users the intricate optimizations done at the low-level (e.g., asynchronous/bulk I/O) but exposes enough parameters if it were necessary to fine-tune the performance. As suggested in the theoretical design, the bitmap is represented as a bit array. Each round of MaNIS is implemented as a series of external-memory sort, using the sort routine that comes with STXXL. In this case, we resort to sorting as the algorithmic description suggested because disk accesses are slower than RAM accesses and random read/writes, as in the parallel case, would be costly (more costly than sorting).

9.2 Evaluation

We empirically investigate the performance of the proposed algorithm. We implemented two variants of the algorithm, one optimized for the multicore architecture and the other optimized for disk-based computation. The algorithms process the graph with exactly the same sequence of buckets and the same MaNIS steps. When started with the same random seed they therefore return the same result. The only difference is in how they perform the communication in the bucketing and each MaNIS step. For a setting of ε, all implementations adopt the convention that the solution produced is no worse than $(1+\varepsilon)(1+\ln n)$.

9.2.1 Experimental Setup

Datasets. Our study uses a diverse collection of instances derived from various sources of popular graphs. Many of the datasets here are obtained from the datasets made publicly available by the Laboratory for Web Algorithmics at Università degli studi di Milano [BRSV11, BV04a]. These datasets are derived from directed graphs of various kinds in a natural way: each node v gives rise to a set and all nodes that v points to are members of the set corresponding to v. We give a detailed description of each instance below and present a summary of these instances in Table 9.1.

Dataset	# of sets	# of elts.	# of edges	avg $\lvert S\rvert$	max $\lvert S\rvert$	Δ
webdocs	1,692,082	5,267,656	299,887,139	177.2	71,472	1,429,525
livejournal-2008	4,817,634	5,363,260	79,023,142	16.4	2,469	19,409
twitter-2010	40,103,281	41,652,230	1,468,365,182	36.6	2,997,469	770,155
twitter-2009	54,127,587	62,539,895	1,837,645,451	34.0	2,968,120	748,285
uk-union	121,503,286	133,633,040	5,507,679,822	45.3	22,429	6,010,077
altavista-2002-nd	532,261,574	1,413,511,386	4,226,882,364	7.9	2,064	299,007

Table 9.1: A summary of the datasets used in our experiments, showing for every dataset the number of sets, the number of elements, the number of edges, the average set size (avg $|S|$), the maximum set size (max $|S|$), and the maximum number of sets containing an element ($\Delta := \max|\{S \ni e\}|$).

—*livejournal-2008* is derived from user-user relationships on the LiveJournal blogging and virtual community site. Our dataset is a snapshot taken by Chierichetti et al. [CKL+09], where each set S is a user of the site and covers all users that are listed as S's friends.
— *webdocs* is a collection of web pages with directed links between them [Goe]. We derive a set system from this graph as described earlier.

— *twitter-2010* is derived from a snapshot taken in 2010 of the follower relationship graph on the popular Twitter network, where there is an edge from x to y if y "follows" x [KLPM10]. The set system is derived using the method described earlier.
— *twitter-2009* is an older, but larger, Twitter snapshot taken in 2009 by a different research group [KMF11].
— *altavista-2002-nd* is the AltaVista web links dataset from 2002 provided by Yahoo! WebScope. The dataset has been preprocessed to remove dangling nodes, as suggested by experts familiar with this dataset[2].
— *uk-union* combines snapshots of webpages in the `.uk` domain taken over a 12-month period between June 2006 and May 2007 [BSV08].

9.2.2 Performance of the Parallel Implementation

The first set of experiments is concerned with the performance of our multicore-optimized program in comparison to existing sequential algorithms. These experiments are designed to test the parallel implementation on the following important metrics:

1. **Solution's Quality**. The parallel algorithm should deliver solutions with no significant loss in quality when compared to the sequential counterpart;
2. **Parallel Overhead.** The parallel algorithm running on a single core should not take much longer than its sequential counterpart, showing empirically that it is work efficient; and
3. **Parallel Speedup**. The parallel algorithm should achieve good speedup[3],indicating that the algorithm can successfully take advantage of parallelism.

The baseline for the experiments is our own implementation of Cormode et al.'s disk-friendly greedy (DFG) algorithm [CKW10]. This is a good baseline for this experiment because DFG achieves significant performance improvements over the standard greedy algorithm by making a geometric-scale bucketing approximation similar to ours. As previously shown, this approximation does not harm the solutions' quality in practice but makes it run much faster on both disk- and RAM- based environments. Our implementation of DFG closely follows the description in their paper but is further optimized for performance. Because of the fine tuning we made to the code, our implementation runs significantly faster than the numbers reported in Cormode et al. on RAM, taking in account the differences between machines. For this reason, we believe our DFG code is a reasonable baseline. We also implemented the

[2]See, e.g., `http://law.dsi.unimi.it/webdata/altavista-2002-nd/`
[3]This measures how much faster it is running on many cores than running sequentially.

standard greedy algorithm for comparison.

Evaluation Setup

Our parallel experiments were performed on a 48-core AMD machine, consisting of *four* Opteron 6168 chips running at 1.9 Ghz. The machine is equipped with 64 GBytes of RAM, running Linux 2.6.35 (Ubuntu 10.10). We compiled our programs with Intel Cilk++ build 8503 using the optimization flag `-O3` [int10a]. The Cilk++ platform, in which the runtime system relies on a work-stealing scheduler, is known to impose only little overhead on both parallel and sequential code.

Dataset	**Standard Greedy**		**DFG**		**Parallel MaNIS**		
	T_1 (sec)	# sets	T_1 (sec)	# sets	T_1 (sec)	T_{48} (sec)	# sets
webdocs	55.35	406,399	9.53	406,340	9.88	4.11	406,367
livejournal-2008	15.19	1,120,594	6.92	1,120,543	15.99	2.91	1,120,599
twitter-2010	584.87	3,846,209	113.08	3,845,345	159.39	21.53	3,845,089
twitter-2009	1,136.2	5,518,039	186.83	5,516,959	222.28	20.84	5,517,864
uk-union	-	-	238.06	18,388,007	422.54	45.47	18,379,547
altavista-2002-nd	-	-	397.60	33,103,284	1,002.50	56.99	33,090,726

Table 9.2: Performance with $\varepsilon = 0.01$ of RAM-based algorithms: the standard greedy implementation, the disk-friendly greedy (DFG) algorithm of Cormode et al., and our MaNIS-based parallel implementation. We show the running time on p cores, denoted by T_p (**in seconds**), and the number of sets in the solutions.

Results

Table 9.2 shows the performance of the three aforementioned RAM-based algorithms when run with $\varepsilon = 0.01$. Several things are clear. *First, parallel MaNIS achieves essentially the same solutions' quality as both DFG and the baseline algorithm.* In fact, with ε set to 0.01 for both DFG and parallel MaNIS, all algorithms produce solutions of roughly the same quality—within about 1% of each other. Interestingly, the standard greedy algorithm does not always yield the best solution. Our experience has been that the additional randomness that parallel MaNIS adds to the greedy algorithm often helps gain better solutions. In a number of datasets above, parallel MaNIS *does* yield the best-quality solutions.

Second, the overhead in running parallel MaNIS is small—under 1.8*x in all but two cases.* This means that parallel MaNIS is likely to be faster than DFG even on a modest number of processors. As the numbers show, except for *livejournal-2008* and *altavista-2002-nd*, parallel MaNIS is at most 1.8x slower than DFG when running on 1 core. We believe this stems from the fact that the number of MaNIS rounds is usually small, so when running sequentially, parallel MaNIS performs more or less the same number of steps as DFG. However, *livejournal-2008* and *altavista-2002-nd* exercise MaNIS more than usual: we have found that in other datasets, at most 5% of the buckets require more than 2 MaNIS rounds, whereas in *livejournal-2008*, 13% of the buckets need 3 or more rounds and in *altavista-2002-nd*, 67% of the buckets need 3 or more rounds—with one bucket running 218 rounds of MaNIS.

Third but perhaps most importantly, parallel MaNIS shows substantial speedups on all but the small datasets. The experiments show that MaNIS achieves upto 17.6x speedup with the speedup numbers ranging between 7.4x and 17.6x—except for the two smallest datasets *webdocs* and *livejournal-2008* which obtain only 2.4x and 5.5x speedup. This shows that the algorithm is able to effectively utilize available cores except when the datasets are too small to take full advantage of parallelism.

To further understand the effects of the number of cores, we study the performance of parallel MaNIS on *altavista-2002-nd* as the number of cores used is varied between 1 and 48 (all cores). As Figure 9.1 shows, the running-time performance of our algorithm scales well with the number of cores until at least 24 cores. After that, even though the performance continues to improve, the marginal benefit diminishes. We believe this is due to saturation of the memory-bandwidth. On newer machines with larger memory bandwidth, we expect to see even better speedups.

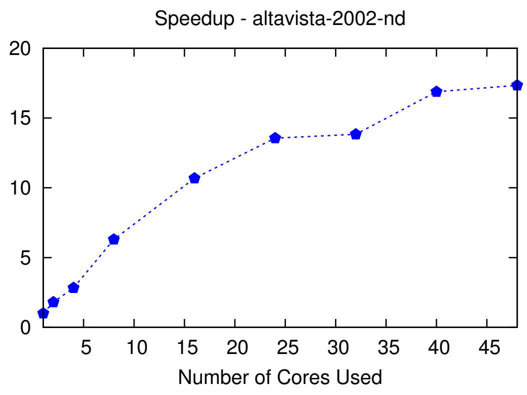


Figure 9.1: Speedup of the parallel MaNIS algorithm (i.e., how much faster is running the algorithm on n cores is over running it sequentially) as the number of cores used is varied.

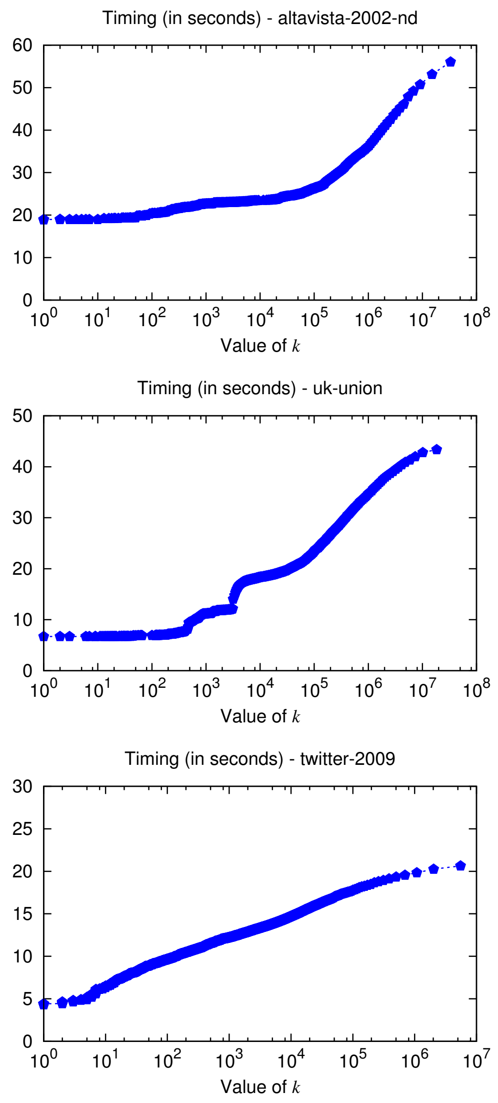


Figure 9.2: Max k-cover performance (**in seconds**) as the value of k is varied.

Max Cover

In the sequential setting, stopping the standard greedy set cover algorithm when it has found k sets gives the optimal $(1 - 1/e)$-approximation to max k-cover. An important feature of the parallel MaNIS algorithm is that it can be stopped early in the same way. To see how one might benefit from stopping the algorithm when the algorithm has found enough sets, we record the time point when k sets are discovered. Presented in Figure 9.2 are plots from our 3 largest datasets (by the number of edges), *altavista-2002-nd*, *uk-union*, and *twitter-2009*. This experiment shows that although the rate varies between datasets, it is clear that most of the sets are added late in the algorithm; therefore, if the value of k of interest is small relative to the set cover solution's size, we can benefit from stopping early, which can often halve the running time. Chierichetti et. al. [CKT10] present results for max k cover on map-reduce but do not report any times, so we were not able to compare.

9.2.3 Performance of Disk-based, Sequential Implementation

The second set of experiments deals with the performance of our disk-optimized implementation in comparison to existing disk-based algorithms. We are interested in evaluating the algorithms on the following metrics: (1) solutions's quality and (2) running time. Since the disk-optimized versions of both DFG and MaNIS yield the same solutions as their parallel counterparts, their relative performance in terms of solutions's quality will be identical to the study conducted earlier for the parallel case. For this reason, in the remaining of this section, we will focus on investigating the running time as well as other performance characteristics of the disk-optimized MaNIS implementation.

Evaluation Setup

Our disk experiments were performed on a 4-core Intel machine although we only make use of a single core running at 2.66 Ghz. The machine is equipped with 8 GBytes of RAM and an Intel X25-M 160 GBytes SSD disk[4] (used both for input and as scratch space). There is a separate magnetic disk which we keep the OS Linux 2.6.38 (Ubuntu 11.04) and other system files. We compiled our programs with *g++* 4.5.2 using the optimization flag `-O3`.

We artificially limited the RAM size available to the set cover process to 512 *MBytes and carefully control all disk-access buffers to use only these* 512 *MBytes.* This may seem unrealistic at first, but this controlled setup models the types of machines available as embedded devices and computing nodes in low-power clusters (e.g., [AFK$^+$09]) and provides a testbed for understanding the performance of these algorithms on such devices.

Table 9.3 reports the performance of the disk-based algorithms when run $\varepsilon = 0.01$. On the larger graphs, the DFG algorithm did not complete within 40 hours. On the smaller sets, the disk based MaNIS is substantially faster (about 4x for *webdocs* and 39.6x for *livejournal*-

[4]Per Intel's specification, it has sustained sequential read and write bandwidths of 250 MB/s and 100 MB/s, resp.

Dataset	**DFG**		**Disk MaNIS**	
	Time	# of sets	Time	# of sets
webdocs	32m50s	406,340	481s	406,367
livejournal-2008	3h5m	1,120,543	280s	1,120,599
twitter-2010	$>$ 40 hrs	-	55m2s	3,845,089
twitter-2009	$>$ 40 hrs	-	1h11m	5,517,864
uk-union	$>$ 40 hrs	-	6h49m	18,379,547
altavista-2002-nd	$>$ 40 hrs	-	13h27m	33,090,726

Table 9.3: Performance with $\varepsilon = 0.01$ of disk-based algorithms: the disk-friendly greedy (DFG) algorithm of Cormode et al., and our disk-based MaNIS implementation. We show the running time and the number of sets in the solutions.

2008). In some cases the timing numbers might seem irregular but a closer look at the results reveals patterns that are worthwhile mentioning:

When the bitmap representing the elements does not fully stay in fast memory (i.e., cache or RAM), the number of passes over the bitmap and the bitmap size crucially determine the performance of both algorithms. In DFG, which consults the bitmap every time it considers a set, this is lower-bounded by the number of sets in the output, whereas in disk MaNIS, which batches "requests" to look up the bitmap and reduces them to one pass over it per MaNIS round, this number is the total number of MaNIS rounds summed across all buckets. This explains the difference in running-time patterns between the two algorithms on *webdocs* and *livejournal-2008*. For this reason also, the two Twitter datasets take roughly the same time to run disk MaNIS despite significant differences in the number of sets outputted.

9.2.4 The Effects of Accuracy Parameter ε

In theory, the dependence on ε in the work bound is $O(1/\log^3(1+\varepsilon))$, which for small ε is roughly $O(1/\varepsilon^3)$. This seems alarming because as we decrease ε (i.e., increase accuracy), $\frac{1}{\varepsilon^3}$ grows rather rapidly, rendering the algorithm unusable in no time; however, in practice, the situation is much better. As we decrease ε, we will observe more buckets but an even larger fraction of these buckets will be empty, reducing the efforts needed to run MaNIS to process them and counteracting the increase in the number of buckets. Table 9.4 shows the effects of ε on *webdocs* for both the disk-optimized and parallel versions.

The numbers show that increasing the accuracy from $\varepsilon = 0.1$ to 0.001 (2 more digits of accuracy) increases the running time by less than 3x. This trend seems to generalize across

Algorithm	$\varepsilon = 0.1$	0.05	0.01	0.001
Disk MaNIS	271	310	481	891
Parallel MaNIS	2.43	2.65	4.11	7.66
# of sets	406,431	406,389	406,367	406,364

Table 9.4: Performance of MaNIS-based algorithms on *webdocs* (**in seconds**) as ε is varied.

the datasets we have. More interesting, however, is the observation that ε only has small effect on the solutions' quality. Our experience has been that with ε sufficiently small, we benefit more from spending the computation time on different random choices than adjusting ε to increase accuracy.

9.3 Conclusion

We presented a parallel cache-oblivious set-cover algorithm that offers essentially the same approximation guarantees as the standard greedy algorithm. As our main contribution, we implemented slight variants of the theoretical algorithm optimized for different hardware setups and provided extensive experimental evaluation showing non-trivial speedups over existing algorithms while yielding the same solution's quality.

Chapter 10

Conclusion and Open Problems

In this thesis, we initiated a principled study of efficient parallel approximation algorithms and presented a set of key algorithmic techniques that aid the design, analysis, and implementation of a wide variety of such algorithms. Traditionally, parallelizing an algorithm entails identifying its dependencies structure, so that we know which parts of the algorithm can be run independently in parallel. When an algorithm is made up mostly of independent operations, it is automatically parallel. Unfortunately, most approximation algorithms have complex dependencies that prevent their operations to be executed in parallel.

The bulk of this work can be seen as identifying independent tasks and decoupling dependent tasks to create additional opportunities for parallelism. In general, this involves staging the dependent computations in a way that we can keep their "interference" under control and we have the ability to properly "merge" the outcome of these dependent computations (Their dependencies will surely cause conflict of some sort). An important advantage of working with approximation algorithms is that the answer needs not be exact, a crucial factor that allows these "merges" to be done reasonably efficiently. For example, MaNIS (Chapter 5) offers a way to bring together overlapping sets such that the amount of overlap in the end is small. Similarly, low-diameter decomposition (Chapter 6) relies crucially on the ability to handle overlapping balls that were grown from different starting points.

Our investigation thus far has led to many tantalizing research questions, which we would like to see resolved in the future. In the rest of this chapter, we comment on future directions for some of these questions.

Small-Width Linear and Semidefinite Programs, and Multiplicative Weight-Update Algorithms

As a continuation of our work in Chapter 4, we would like to gain a deeper understanding of what can be solved in parallel in this way. Specifically,

Question 10.1 *What LP and SDP formulations have a small width? Is it possible to develop a general "sparsification" technique to reduce the width of those that are not small enough?*

Learning Linear Separators. Closely related to this theme is the problem of learning a linear separator. Many real-life datasets are known to have large margin. On these datasets, it was observed that classifiers such as linear separators and SVMs can be trained quickly. As a first step in this direction, we ask: *how to efficiently train a linear classifier in parallel if the dataset has large margin?*

In the sequential setting, we have a classic algorithm whose running time is a function of the margin. More specifically, if the margin is γ, then the Perceptron algorithm has running time $O(\frac{1}{\gamma^2}nd)$, which can be parallelized to get a $O(\frac{1}{\gamma^2}nd)$-work and $O(\frac{1}{\gamma^2}\log n \log d)$-depth algorithm. Thus, the parallel algorithm is work efficient with respect to the sequential counterpart. In this case, even though there is a considerable amount of parallelism, the depth is polynomial when $1/\gamma^2 = \Omega(n)$. *It is natural to wonder whether the depth can be improved to a quantity that depends on, e.g.,* $\log(1/\gamma)$ *as opposed to* $1/\gamma^{O(1)}$. To this end, we are able to show that when the margin γ is exponentially small in n, such an improvement is unlikely.

Theorem 10.2 *The problem of finding a linear separator when* $\gamma \leq \frac{1}{16n}4^{-n}$ *is* P*-hard.*

MaNIS Extensions

Our formulation of MaNIS assumes set-based options, where the interference is additive (set union and intersection). This can be seen as requiring the goodness measure of the options to be a unimodular function (in this case, the set cardinality function).

Question 10.3 *Is there an analog of MaNIS for submodular functions? Could this generalization be used to solve problems like maximizing a submodular function subject to a budget constraint?*

Low-Stretch Spanning Trees and SDD Solvers

In this work, we presented a near linear-work parallel algorithm for graph decomposition with strong-diameter guarantees and parallel algorithms for constructing $2^{O(\sqrt{\log n \log\log n})}$-stretch spanning trees and $O(\log^{O(1)} n)$-stretch ultra-sparse subgraphs. The ultra-sparse subgraphs were shown to be useful in the design of a near linear-work parallel SDD solver.

An interesting avenue of research is to design a (near) linear-work parallel algorithm for constructing a low-stretch tree with polylogarithmic stretch. Currently, we only know how to get ultrasparse graphs if polylogarithmic stretch is required.

Question 10.4 *Is it possible to construct a low-stretch spanning tree with polylog stretch in near linear-work and low-depth?*

Since we do not know of an efficient low-depth single-source shortest-path algorithm, a good starting point for this question is perhaps exploring whether we can build a low-stretch spanning tree with polylog stretch in a bottom-up manner. All of the constructions thus far that yield polylog stretch are top-down (á la the FRT-tree construction), which requires growing balls out to large radii.

Another tantalizing problem is that of devising a (near) work-efficient $O(\log^{O(1)} n)$-depth SDD solver. Such a solution will lead to immediate significant improvements to many parallel algorithms. We believe radical changes to the solver framework will be necessary.

The Complexity of k-Median and k-Means

Our work shows RNC constant approximation algorithms for k-median and k-means for small k (specifically, for $k \leq \text{polylog}(n)$). For large k, we only know of an RNC $O(\log k)$-approximation.

Question 10.5 *Is it possible to derive* RNC constant-approximation algorithms for k-median and k-means for *any* k? Or, can we show that this is P-complete?

Bibliography

[AA97] Baruch Awerbuch and Yossi Azar. Buy-at-bulk network design. In *Proceedings of the 38th Symposium on the Foundations of Computer Science (FOCS)*, pages 542–547, 1997. 126

[ABB02] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3), 2002. 18

[ABN08] Ittai Abraham, Yair Bartal, and Ofer Neiman. Nearly tight low stretch spanning trees. In *FOCS*, pages 781–790, 2008. 89

[ACST06] Haim Avron, Doron Chen, Gil Shklarski, and Sivan Toledo. Combinatorial preconditioners for scalar elliptic finite-elements problems. *under submission*, 2006. 140

[AFK+09] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. Best paper award. 166

[AGK+04] Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. Local search heuristics for k-median and facility location problems. *SIAM J. Comput.*, 33(3):544–562, 2004. 6, 7, 25, 43, 46, 123

[AK07] Sanjeev Arora and Satyen Kale. A combinatorial, primal-dual approach to semidefinite programs. In *STOC*, pages 227–236, 2007. 8, 50, 51, 52, 53, 57

[AKPW95] Noga Alon, Richard M. Karp, David Peleg, and Douglas West. A graph-theoretic game and its application to the k-server problem. *SIAM J. Comput.*, 24(1):78–100, 1995. 10, 89, 90, 96

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. Assoc. Comput. Mach.*, 32(4):804–823, 1985. 10, 89

[Axe94] Owe Axelsson. *Iterative Solution Methods*. Cambridge University Press, New York, NY, 1994. 143

[Bar98] Yair Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 30th ACM Symposium on the Theory of Computing (STOC)*, pages 161–168, 1998. 11, 113, 123

[BBK03] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *SODA*, pages 679–688, 2003. 5, 20, 152

[BBK04] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. An experimental analysis of a compact graph representation. In *ALENEX/ANALC*, pages 49–61, 2004. 5, 20, 134

[BCG$^+$08] Guy E. Blelloch, Rezaul Alam Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA*, pages 501–510, 2008. 134, 138, 140

[BDK$^+$08] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34(4):1–22, 2008. 133, 135, 141

[BFF$^+$09] Aydin Buluc, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA*, pages 233–244, 2009. 135

[BFGS11] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*, pages 355–366, 2011. 18, 19, 24, 156, 157

[BG04] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA'04*, 2004. 18

[BGK$^+$11] Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan. Near linear-work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. In *SPAA*, pages 13–22, 2011. 14

[BGS10] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious sorting. In *SPAA'10*, 2010. 18, 24, 26, 134

[BH03] Erik G. Boman and Bruce Hendrickson. Support theory for preconditioning. *SIAM J. Matrix Anal. Appl.*, 25(3):694–717, 2003. 140, 143

[BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial: second edition.* Society for Industrial and Applied Mathematics, 2000. 133, 141

[BHV04] Erik G. Boman, Bruce Hendrickson, and Stephen A. Vavasis. Solving elliptic finite element systems in near-linear time with support preconditioners. *CoRR*, cs.NA/0407022, 2004. 140

[BKMT10] Guy E. Blelloch, Ioannis Koutis, Gary L. Miller, and Kanat Tangwongsan. Hierarchical diagonal blocking and precision reduction applied to combinatorial multigrid. In *Supercomputing*, 2010. 12, 14

[BKTW07] Michael A. Bender, Bradley C. Kuszmaul, Shang-Hua Teng, and Kebin Wang. Optimal cache-oblivious mesh layout. Computing Research Repository (CoRR) abs/0705.1033, 2007. 134

[BM98] Guy E. Blelloch and Bruce M. Maggs. *Handbook of Algorithms and Theory of Computation*, chapter Parallel Algorithms. CRC Press, Boca Raton, FL, 1998. 62

[BNBH$^+$98] Amotz Bar-Noy, Mihir Bellare, Magnús M. Halldórsson, Hadas Shachnai, and Tami Tamir. On chromatic sums and distributed resource allocation. *Inform. and Comput.*, 140(2):183–202, 1998. 75

[BPT11] Guy E. Blelloch, Richard Peng, and Kanat Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *SPAA*, pages 23–32, 2011. 14, 156, 157, 158, 159

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974. 17

[BRS94] Bonnie Berger, John Rompel, and Peter W. Shor. Efficient *NC* algorithms for set cover with applications to learning and geometry. *J. Comput. Syst. Sci.*, 49(3):454–477, 1994. 3, 8, 21, 60, 64, 156

[BRSV11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011. 161

[BS11] Guy E. Blelloch and Julian Shun. A simple parallel cartesian tree algorithm and its application to suffix tree construction. In *ALENEX*, pages 48–58, 2011. 115, 121

[BSV08] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008. 162

[BT10] Guy E. Blelloch and Kanat Tangwongsan. Parallel approximation algorithms for facility-location problems. In *SPAA*, pages 315–324, 2010. 11, 14, 60, 61, 77, 80, 83, 84, 114, 123, 126, 130

[BV93] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993. 115

[BV04a] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press. 161

[BV04b] S. Boyd and L. Vandenberghe. *Convex Optimization*. Camebridge University Press, 2004. 86

[Byr07] Jaroslaw Byrka. An optimal bifactor approximation algorithm for the metric uncapacitated facility location problem. In *APPROX'07*, 2007. 29–43. 25

[CG05] Moses Charikar and Sudipto Guha. Improved combinatorial algorithms for facility location problems. *SIAM J. Comput.*, 34(4):803–824, 2005. 25

[CGG$^+$95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *SODA*, pages 139–149, 1995. 155

[CGH$^+$05] Julia Chuzhoy, Sudipto Guha, Eran Halperin, Sanjeev Khanna, Guy Kortsarz, Robert Krauthgamer, and Joseph Naor. Asymmetric k-center is $\log^* n$-hard to approximate. *J. ACM*, 52(4):538–551, 2005. 77

[CGTS02] Moses Charikar, Sudipto Guha, Éva Tardos, and David B. Shmoys. A constant-factor approximation algorithm for the k-median problem. *J. Comput. System Sci.*, 65(1):129–149, 2002. Special issue on STOC, 1999 (Atlanta, GA). 25

[Chu98] Fabián A. Chudak. Improved approximation algorithms for uncapacitated facility location. In *Integer programming and combinatorial optimization (IPCO)*, volume 1412 of *Lecture Notes in Comput. Sci.*, pages 180–194. Springer, Berlin, 1998. 25

[Chv79a] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):pp. 233–235, 1979. 59

[Chv79b] V. Chvátal. The tail of the hypergeometric distribution. *Discrete Mathematics*, 25(3):285–287, 1979. 88

[CKL$^+$09] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD*, pages 219–228, 2009. 161

[CKM$^+$11] Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *STOC*, pages 273–282, 2011. 86

[CKT10] Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-cover in map-reduce. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 231–240, New York, NY, USA, 2010. ACM. 61, 74, 166

[CKW10] Graham Cormode, Howard J. Karloff, and Anthony Wirth. Set cover algorithms for very large datasets. In *CIKM*, pages 479–488, 2010. 155, 162

[Coh93] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. In *Proceedings of the 1993 IEEE 34th Annual Foundations of Computer Science*, pages 648–658, Washington, DC, USA, 1993. IEEE Computer Society. 90

[Coh00] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *J. ACM*, 47(1):132–166, 2000. 88

[Col88] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988. 26

[CS03] Fabián A. Chudak and David B. Shmoys. Improved approximation algorithms for the uncapacitated facility location problem. *SIAM J. Comput.*, 33(1):1–25, 2003. 25

[Dav94] Timothy A. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 92, 1994. `http://www.cise.ufl.edu/research/sparse/matrices/`. 145

[DKS08] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: standard template library for xxl data sets. *Software: Practice and Experience*, 38(6):589–637, 2008. 160

[DS08] Samuel I. Daitch and Daniel A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 451–460, 2008. 86, 140

[EEST05] Michael Elkin, Yuval Emek, Daniel A. Spielman, and Shang-Hua Teng. Lower-stretch spanning trees. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 494–503, New York, NY, USA, 2005. ACM Press. 89

[Fei98] U. Feige. A threshold of $\ln n$ for approximating set cover. *J. ACM*, 45(4):634–652, 1998. 59

[FKL$^+$91] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel Dominic Sleator, and Neal E. Young. Competitive paging algorithms. *J. Algorithms*, 12(4):685–699, 1991. 18

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298, 1999. 17, 134, 156

[FLT04] Uriel Feige, László Lovász, and Prasad Tetali. Approximating min sum set cover. *Algorithmica*, 40(4):219–234, 2004. 75, 76

[FRT04] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *J. Comput. System Sci.*, 69(3):485–497, 2004. 11, 113, 114, 115, 117, 118, 123

[GHR95] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995. 78

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Company, San Francisco, 1979. 49

[GK99] Sudipto Guha and Samir Khuller. Greedy strikes back: improved facility location algorithms. *J. Algorithms*, 31(1):228–248, 1999. 25

[GL81] A George and J. W. Liu. *Computer Solutions of Large SParse Positive Definite Systems*. Prentice-Hall, 1981. 134

[GLS06] Joachim Gehweiler, Christiane Lammersen, and Christian Sohler. A distributed $O(1)$-approximation algorithm for the uniform facility location problem. In *SPAA'06*, pages 237–243, 2006. 23

[Goe] B. Goethals. Frequent itemset mining dataset repository. http://fimi.ua.ac.be/data/. 161

[Gon85] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoret. Comput. Sci.*, 38(2-3):293–306, 1985. 25, 77

[Gra06] Leo Grady. Random walks for image segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 2(11):1768–1783, 2006. 141

[Gra08] Leo Grady. A lattice-preserving multigrid method for solving the inhomogeneous poisson equations used in image analysis. In David A. Forsyth, Philip H. S. Torr, and Andrew Zisserman, editors, *ECCV (2)*, volume 5303 of *Lecture Notes in Computer Science*, pages 252–264. Springer, 2008. 141

[Gre96] Keith Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, October 1996. CMU CS Tech Report CMU-CS-96-123. 87, 102, 140

[GT08] Anupam Gupta and Kanat Tangwongsan. Simpler analyses of local search algorithms for facility location. *CoRR*, abs/0809.2554, 2008. 6, 7, 14, 25, 43, 44

[GVL96] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Press, 3rd edition, 1996. 104

[GW95] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. Assoc. Comput. Mach.*, 42(6):1115–1145, 1995. 4, 7, 49

[HL95] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings ACM/IEEE conference on Supercomputing*, page 28, 1995. 134

[Hoe63] Wassily Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. 88

[HS85] Dorit S. Hochbaum and David B. Shmoys. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985. 6, 7, 24, 40, 77

[HS86] Dorit S. Hochbaum and David B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. *J. Assoc. Comput. Mach.*, 33(3):533–550, 1986. 25

[int10a] Intel + Cilk , 2010. 144, 163

[int10b] Intel Math Kernel Library , 2010. 144

[IYV04] Eun-Jin Im, Katherine A. Yelick, and Richard Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004. 133, 135

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. 16, 17, 26, 30, 78, 100, 115, 122

[JMM$^+$03] Kamal Jain, Mohammad Mahdian, Evangelos Markakis, Amin Saberi, and Vijay V. Vazirani. Greedy facility location algorithms analyzed using dual fitting with factor-revealing LP. *Journal of the ACM*, 50(6):795–824, 2003. 6, 24, 25, 29, 30, 32, 61, 80, 83

[Joh74] David S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974. 59

[JV01a] Kamal Jain and Vijay Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and Lagrangean relaxation. *Journal of the ACM*, 48(2):274–296, 2001. (Preliminary version in *40th FOCS*, pages 2–13, 1999). 6

[JV01b] Kamal Jain and Vijay V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and Lagrangian relaxation. *Journal of the ACM*, 48(2):274–296, 2001. 6, 24, 25, 35

[JY11] Rahul Jain and Penghui Yao. A parallel approximation algorithm for positive semidefinite programming. *CoRR*, abs/1104.2502, 2011. 22

[Kar72] R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. 59

[Kha04] Rohit Khandekar. *Lagrangian Relaxation Based Algorithms for Convex Programming Problems*. PhD thesis, Indian Institute of Technology Delhi, 2004. 51

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998. 134, 152

[KL96] Philip N. Klein and Hsueh-I Lu. Efficient approximation algorithms for semidefinite programs arising from MAX CUT and COLORING. In *STOC*, pages 338–347, 1996. 50

[KLPM10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010. 162

[KM07] Ioannis Koutis and Gary L. Miller. A linear work, $O(n^{1/6})$ time, parallel algorithm for solving planar laplacians. In *SODA*, pages 1002–1011, 2007. 85, 102

[KM08] Ioannis Koutis and Gary L. Miller. Graph partitioning into isolated, high conductance clusters: Theory, computation and applications to preconditioning. In *Symposiun on Parallel Algorithms and Architectures (SPAA)*, 2008. 140, 150

[KM09] Ioannis Koutis and Gary Miller. The Combinatorial Multigrid Solver. In *The 14th Copper Mountain Conference on Multigrid Methods*, March 2009. Conference Talk. 140

[KMF11] U. Kang, Brendan Meeder, and Christos Faloutsos. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *PAKDD (2)*, pages 13–25, 2011. 162

[KMN$^+$04] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. A local search approximation algorithm for k-means clustering. *Comput. Geom.*, 28(2-3):89–112, 2004. 25, 44

[KMP10] Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD linear systems. In *FOCS*, pages 235–244, 2010. 10, 89, 102, 103, 105, 140

[KMP11] Ioannis Koutis, Gary L. Miller, and Richard Peng. Solving sdd linear systems in time $\tilde{O}(m \log n \log(1/\epsilon))$. *CoRR*, abs/1102.4842, 2011. 85

[KMT09] Ioannis Koutis, Gary L. Miller, and David Tolliver. Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing. In *International Symposium of Visual Computing*, pages 1067–1078, 2009. 140, 141

[Kou07] Ioannis Koutis. *Combinatorial and algebraic algorithms for optimal multilevel algorithms*. PhD thesis, Carnegie Mellon University, Pittsburgh, May 2007. CMU CS Tech Report CMU-CS-07-131. 105, 140, 142

[KPR00] Madhukar R. Korupolu, C. Greg Plaxton, and Rajmohan Rajaraman. Analysis of a local search heuristic for facility location problems. *J. Algorithms*, 37(1):146–188, 2000. (Preliminary version in 9th SODA, 1998). 25

[KS97] Philip N. Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, 1997. 88

[KVY94] Samir Khuller, Uzi Vishkin, and Neal E. Young. A primal-dual parallel approximation technique applied to weighted set and vertex covers. *J. Algorithms*, 17(2):280–289, 1994. 3, 60

[KW85] Richard M. Karp and Avi Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM*, 32(4):762–773, 1985. 64

[KY09] Christos Koufogiannakis and Neal E. Young. Distributed and parallel algorithms for weighted vertex cover and other covering problems. In *PODC*, pages 171–179, 2009. 3, 21

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. 30, 100

[LN93] Michael Luby and Noam Nisan. A parallel approximation algorithm for positive linear programming. In *STOC'93*, pages 448–457, New York, NY, USA, 1993. 3, 22

[LT79] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979. 20

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986. 28, 64

[McC07] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/. xiii, 145

[MHDY09] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine A. Yelick. Minimizing communication in sparse matrix solvers. In *SC*. ACM, 2009. 133, 136

[MMSV01] Mohammad Mahdian, Evangelos Markakis, Amin Saberi, and Vijay Vazirani. A greedy facility location algorithm analyzed using dual fitting. In *Approximation, randomization, and combinatorial optimization (Berkeley, CA, 2001)*, volume 2129 of *Lecture Notes in Comput. Sci.*, pages 127–137. Springer, Berlin, 2001. 25

[MN08] Adrian C. Muresan and Yvan Notay. Analysis of aggregation-based multigrid. *SIAM J. Scientific Computing*, 30(2):1082–1103, 2008. 141

[MP04] Ramgopal R. Mettu and C. Greg Plaxton. Optimal time bounds for approximate clustering. *Machine Learning*, 56(1-3):35–60, 2004. 125

[MR89] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. In Silvio Micali, editor, *Randomness and Computation*, pages 47–72. JAI Press, Greenwich, Connecticut, 1989. Vol. 5. 105

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995. 34

[MRK88] Gary L. Miller, Vijaya Ramachandran, and Erich Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Comput.*, 17(4):687–695, 1988. 115

[MTV91] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *FOCS*, pages 538–547, 1991. 20

[MW05] Thomas Moscibroda and Roger Wattenhofer. Facility location: distributed approximation. In *PODC'05*, pages 108–117, 2005. 23

[MYZ02] Mohammad Mahdian, Yinyu Ye, and Jiawei Zhang. Improved approximation algorithms for metric facility location problems. In *Approximation algorithms for combinatorial optimization*, volume 2462 of *Lecture Notes in Comput. Sci.*, pages 229–242. Springer, Berlin, 2002. 25

[OLHB02] Leonid Oliker, Xiaoye Li, Parry Husbands, and Rupak Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Rev.*, 44(3):373–393, 2002. 133, 134

[OW08] Ryan O'Donnell and Yi Wu. An optimal SDP algorithm for Max-Cut, and equally optimal long code tests. In *STOC*, pages 335–344, 2008. 50

[PF90] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw.*, 16(4):303–324, 1990. 133, 134

[PP09] Saurav Pandit and Sriram V. Pemmaraju. Return of the primal-dual: distributed metric facility location. In *PODC'09*, pages 180–189, 2009. 23, 36

[PST95] Serge A. Plotkin, David B. Shmoys, and Éva Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 20(2):257–301, 1995. 51

[PT03] Martin Pál and Éva Tardos. Group strategyproof mechanisms via primal-dual algorithms. In *FOCS'03*, pages 584–593, 2003. 25

[PV98] Rina Panigrahy and Sundar Vishwanathan. An $O(\log^* n)$ approximation algorithm for the asymmetric *p*-center problem. *J. Algorithms*, 27(2):259–268, 1998. 61, 73, 77, 78

[Ren01] James Renegar. *A mathematical view of interior-point methods in convex optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001. 86

[RR89] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989. 62, 71

[RV98] Sridhar Rajagopalan and Vijay V. Vazirani. Primal-dual *RNC* approximation algorithms for set cover and covering integer programs. *SIAM J. Comput.*, 28(2):525–540, 1998. 3, 8, 21, 33, 60, 64, 71, 72, 156

[Saa90] Y. Saad. Sparskit: A basic took kit for sparse matrix computations. Technical Report 90-20, RIACS, NASA Ames Research Center, 1990. 133, 135

[Sei92] Raimund Seidel. Backwards analysis of randomized geometric algorithms. In *Trends in Discrete and Computational Geometry, volume 10 of Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, 1992. 122

[Sha48] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27, 1948. 124, 128

[Ska09] Matthew Skala. Hypergeometric tail inequalities: ending the insanity, 2009. 88

[Spi10] Daniel A. Spielman. Algorithms, Graph Theory, and Linear Equations in Laplacian Matrices. In *Proceedings of the International Congress of Mathematicians*, 2010. 85

[Sri01] Aravind Srinivasan. New approaches to covering and packing problems. In *SODA*, pages 567–576, 2001. 3

[SS08] Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. In *STOC*, pages 563–568, 2008. 86

[SSP07] Johannes Singler, Peter Sanders, and Felix Putze. The multi-core standard template library. In *Euro-Par*, pages 682–694, 2007. 157, 160

[ST03] Daniel A. Spielman and Shang-Hua Teng. Solving sparse, symmetric, diagonally-dominant linear systems in time $O(m^{1.31})$. In *FOCS*, pages 416–427, 2003. 102, 105

[ST04] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 81–90, June 2004. 140

[ST06] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for pre-conditioning and solving symmetric, diagonally dominant linear systems. *CoRR*, abs/cs/0607105, 2006. 10, 89, 102, 105, 106, 108

[STA97] David B. Shmoys, Éva Tardos, and Karen Aardal. Approximation algorithms for facility location problems (extended abstract). In *STOC*, pages 265–274, 1997. 6, 7, 25, 40

[SV88] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988. 115

[Svi02] Maxim Sviridenko. An improved approximation algorithm for the metric uncapacitated facility location problem. In *Integer programming and combinatorial optimization*, volume 2337 of *Lecture Notes in Comput. Sci.*, pages 240–257. Springer, Berlin, 2002. 25

[Tam96] Arie Tamir. An $O(pn^2)$ algorithm for the p-median and related problems on tree graphs. *Operations Research Letters*, 19(2):59–64, 1996. 123

[Ten10] Shang-Hua Teng. The Laplacian Paradigm: Emerging Algorithms for Massive Graphs. In *Theory and Applications of Models of Computation*, pages 2–14, 2010. 85

[TKI$^+$08] D. A. Tolliver, I. Koutis, H. Ishikawa, J. S. Schuman, and G. L. Miller. Automatic multiple retinal layer segmentation in spectral domain oct scans via spectral rounding. In *ARVO Annual Meeting*, May 2008. 141

[TM06] David Tolliver and Gary L. Miller. Graph partitioning by spectral rounding: Applications in image segmentation and clustering. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2006)*, pages 1053–1060, 2006. 141

[Tol97] Sivan Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *IBM Journal of Research and Development*, 1997. 133, 135

[Tre01] Luca Trevisan. Non-approximability results for optimization problems on bounded degree instances. In *STOC*, pages 453–461, 2001. 8, 55

[TSO00] Ulrich Trottenberg, Anton Schuller, and Cornelis Oosterlee. *Multigrid*. Academic Press, 1st edition, 2000. 133, 141

[UY91] Jeffrey D. Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM J. Comput.*, 20(1):100–125, 1991. 88

[Vaz01] Vijay V. Vazirani. *Approximation algorithms*. Springer-Verlag, Berlin, 2001. 71, 72

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2), 2001. 156

[WC90] Qingzhou Wang and Kam Hoi Cheng. Parallel time complexity of a heuristic algorithm for the k-center problem with usage weights. In *Proc. IEEE Symposium on Parallel and Distributed Processing*, pages 254 –257, December 1990. 3, 23

[WL06] Jeremiah Willcock and Andrew Lumsdaine. Accelerating sparse matrix computations via data compression. In *ICS*, pages 307–316, 2006. 133, 135

[WOV$^+$07a] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, 2007. 12

[WOV$^+$07b] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, 2007. 133, 135, 138

[Ye97] Y. Ye. *Interior point algorithms: theory and analysis*. Wiley, 1997. 86

[You95] Neal E. Young. Randomized rounding without solving the linear program. In *SODA*, pages 170–178, 1995. 73

[You01] Neal E. Young. Sequential and parallel algorithms for mixed packing and covering. In *FOCS*, pages 538–546, 2001. 3, 22