On Local Representations of Graphs and Networks

by

Lenore Jennifer Cowen

B.A., Yale University (1987)

Submitted to the Department of Applied Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1993

© Massachusetts Institute of Technology

The author hereby grants to MIT permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Signature of Author ....
Department of Applied Mathematics
May 1, 1993

Certified by ....
Daniel J. Kleitman
Professor of Applied Mathematics
Thesis Supervisor

Accepted by ....
Alar Toomre
Chair, Department of Applied Mathematics

ARCHIVES

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUL 27 1993

LIBRARIES

On Local Representations of Graphs and Networks

by

Lenore Jennifer Cowen

Submitted to the Department of Applied Mathematics

on May 1, 1993, in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

Abstract

Variations of *network decomposition,* a decomposition of a graph or network into low-diameter neighborhoods, are constructed in the parallel, distributed, and sequential models of computation. Chapter 1 gives background and definitions, and it surveys new and known results. Chapter 2 obtains an NC algorithm for a coloring version of a network decomposition, which we call weak network decomposition. It does this by modifying an algorithm of Linial and Saks to depend on only pairwise independence, and then removing randomness. Chapter 3 obtains sublinear time deterministic distributed algorithms for all alternative notions of network decomposition. Chapter 4 presents a sequential near-linear construction of layered sparse neighborhood covers, with applications to approximating all-pairs shortest paths.

The network decomposition structure was first introduced by Awerbuch, Goldberg, Luby, and Plotkin as a means to deterministically construct a maximal independent set in a distributed network. In Chapter 5, we return to the MIS problem in distributed computing and present a highly robust randomized protocol that constructs a maximal independent set in an entirely asynchronous environment in $O(\log n)$ expected time. Applications include an optimal solution to the generalized dining philosophers problem, where a process that competes with d other unit-time processes for resources it needs exclusively to execute, is scheduled in $O(d)$ expected time.

The constructions presented here can also be seen in a broader context as the beginning of an investigation of what functions of graphs and networks can be computed locally. Since locality can be key to designing efficient algorithms, to increase what we know how to compute without complete global knowledge of graph or network topology is important, and we suggest some open questions and future directions, in the conclusion.

Thesis Supervisor: Daniel J. Kleitman

Title: Professor of Applied Mathematics

Contents

Acknowledgments **9**

Introduction **13**

1 Preliminaries and Definitions **17**

1.1 Introductory definitions 17

1.2 Network decomposition 18

1.2.1 Weak network decomposition 18

1.2.2 The simple greedy algorithm. 19

1.2.3 Strong network decomposition 20

1.2.4 The cover definitions 21

1.2.5 A composite structure 22

1.3 Extension to separation and local neighborhoods 23

1.4 Summary of results 24

2 An NC algorithm for graph decomposition **27**

2.1 Introduction 27

2.2 An *RNC* algorithm 28

2.2.1 The *RNC* algorithm of Linial-Saks 28

2.2.2 Overview of the pairwise independent *RNC* algorithm 29

2.2.3 The *RNC* algorithm 30

2.2.4 Analysis of the algorithm's performance 31

2.3 The *NC* algorithm 34

2.3.1 The pairwise independent distribution 34

2.3.2 Searching the sample space 35

3 Distributed Network Decomposition 39

3.1 Introduction 39

3.1.1 Previous work 39

3.1.2 Structure of the chapter: 41

3.2 Linial's model 41

3.3 Weak diameter network decomposition 42

3.3.1 Algorithm `Color` 42

3.3.2 Analysis of `Color` 45

3.4 The transformer algorithm 46

3.4.1 Algorithm `Transform` 47

3.4.2 The modified procedure `Cover` 48

3.4.3 The resulting algorithms 50

4 Fast Construction of Sparse Neighborhood Covers 51

4.1 Introduction 51

4.1.1 Structure of this chapter 53

4.2 Overview of the construction 53

4.2.1 Growth of a single set 54

4.2.2 When to stop 55

4.2.3 How to grow around existing sets 56

4.3 Formal description of the algorithm 58

4.3.1 Procedure `Cluster` 59

4.3.2 Procedure `Cover` 59

4.3.3 Covering all neighborhoods 60

4.4 Analysis of the algorithm 61

4.4.1 Correctness 61

4.4.2 Complexity analysis 62

4.5 Approximating k-pairs shortest paths 63

4.5.1 Tree covers 65

4.5.2 The algorithm 65

4.5.3 The analysis 66

5 Wait-free Asynchronous MIS **69**

5.1 Introduction 70

5.1.1 Previous results 71

5.1.2 Our results 72

5.1.3 Structure of this chapter 73

5.2 The model and the problem statement 73

5.3 The Protocol 75

5.3.1 Review of Luby's protocol 76

5.3.2 The difficulty of asynchrony 76

5.3.3 The description of the code 76

5.4 Analysis of the algorithm 79

5.5 Dining philosophers 85

5.5.1 History and definitions 85

5.5.2 The reduction from asynchronous MIS 86

Conclusion **89**

Bibliography **91**

Acknowledgments

First, I would like to thank my advisor, Danny J. Kleitman, for his brilliance, help, support, and numerous productive discussions from the mathematical to the practical. I have learned so much from my interaction with him, and I am truly grateful to him for having recognized something in me, long before it was truly developed.

I would also like to thank my collaborators and co-authors, who were wonderful to work with, and for their kind permission to put our joint work into this thesis. Thanks to Baruch Awerbuch, Bonnie Berger and David Peleg, who worked with me on network decomposition, and to Baruch Awerbuch and Mark Smith, who worked with me on asynchronous MIS and the dining philosophers problem.

The members of my thesis committee, Baruch Awerbuch, Bonnie Berger and Tom Leighton, have all substantially enriched my experience at MIT. Baruch Awerbuch has taught me everything I know about distributed computing, and I continue to be impressed by his penetrating insight. Tom Leighton is so often right, about what's interesting and/or tractable, that his nickname around here is "The Oracle"– his insight, taste, advice and support have been invaluable. Finally, very special thanks go to Bonnie Berger, who has been a second advisor to me. Bonnie is to be credited with "discovering" me and teaching me the craft of research beginning when she was a senior graduate student and I was a younger one. She has more talents than even she imagines, and I'm sure will do even greater and greater things in the future. Thanks for being a real friend– stay in touch.

Several people have given good feedback, and helpful comments on earlier drafts of the material in this thesis. Thanks to Dana Sussman for reading the Introduction, John Rompel and Mike Saks for discussions on the material in Chapter 2, Tom Leighton (Chapter 3) and Nancy Lynch (Chapter 5).

Thanks to all the support staff at the Lab for Computer Science, in particular, thanks to Be Hubbard, and David Jones, who manage to stay on top of everything that happens on the third floor of LCS. Many thanks to Maureen Lynch, for spirited help on the administrative end for everything! Special thanks to Phyllis Ruby, for all her help not just to me, but for her special attention to all the women graduate students in the Mathematics department– in the midst of a changing population of students who learn lots of things and then (happily) graduate, Phyllis has remained our memory and our resource, and a wealth of quiet helpful advice to all graduate

students.

I want to thank Shafi Goldwasser for advising me early in my graduate career. Also thanks to Silvio Micali, for his culture, taste, brilliance and philosophy, Gian-Carlo Rota, my favorite philosophy professor, for many stimulating discussions, Charles Leiserson, for just being an all-around, brilliant, great kind of guy, Nancy Lynch, Ron Rivest, Shanghua Teng and Jim Propp, all of whom add immensely to the culture here at MIT. I'd also like to thank David Johnson and all the people at Bell Labs, particularly Joan Feigenbaum, who really seem to take this mentoring business very seriously. The support and interest in young developing researchers I have seen at Bell Labs is without equal, and I feel very lucky to have gotten a chance to meet some of you.

The graduate students here in MIT really form a community– two communities, in fact. I want to thank Bard Bloom, Margaret Fleck, Joe Killian, Marios Papaefthymiou, Bob Sloan, Donna Slonim, Mark Smith, Ellen Spertus, and David Williamson over in the Lab for Computer Science, and Sara Billey, Alan Edelman, Dan Klain, Dan Spielman, and of course the other "Kleitman's Kids": Wayne Goddard, Richard Ehrenborg, and Leonard Shulman, over in the department of Mathematics. Special thanks to Mihir Bellare, Wayne Goddard and John Rompel, for being true colleagues and true friends. Special thanks as well to Lalita Jategaonkar, for being the best officemate through our mutual job search and thesis stress– Lalita, best of luck next year! I know you'll do great.

I want to thank Jill Burger for being a true friend, and helping me keep some perspective while at MIT, Tim Wright for friendship, train schedules, new and novel housekeeping strategies, bridge and great puns, Marilyn Richards for teaching me how to run an organization (plus too many other things to name), Tim Gardner for car thoughts and long talks, and Norman Baker for his unerring faith and support (but also for figuring out the mass mice mystery!) I also want to thank Katya Reinmann, Patrick Yacono, Cliff Schmidt and Michelle Leong, Rob Carter, Julia Vail, Cindy Kogut, Elizabeth Stone, Jonathan Young, Martha Sullivan, Sara Howe, Ted Tso, The Silliman gang: (Tim Wright, Kathy Gisser, Gillian Horvath, Bret Shefter, John Brewer), Spike and Tina.

There were some early figures who helped me discover I liked mathematics, as far back as high school, and into the early years of college. I'd like to thank David Wayne, Carl Goodman, Paul Cinco, Judy Goldsmith, Simon Thomas, Stephen Lempp, Pal Rosza, and especially Dana

Angluin, who is to be credited with introducing me to theoretical computer science when I was an undergraduate at Yale.

Finally I would like to thank my parents, Robert and Ilsa Cowen, my sister Minda and my grandfather Walter, for their love, support, and belief. In particular, I dedicate this thesis to my father, Robert Cowen, who would have been happy with whatever I chose to do with my life, but I think is extra thrilled that I chose to follow in his footsteps and pursue mathematics.

Introduction

Algorithms that exploit locality are often more efficient. If we can design algorithms that are modular with respect to a local representation of a graph or network, these can often be cleaner, simpler, and avoid expensive global computation. In this thesis, we consider two different families of local representations. Chapters 1-4 of this thesis are concerned with variations of the *network decomposition* problem, a data structure that partitions a graph into low-diameter neighborhoods. Typically, low-diameter will mean diameter $O(\log n)$, where n is the number of nodes in the graph. In Chapter 5, we look at only constant diameter neighborhoods, and turn to the problem of local resource allocation.

Network decomposition

Network decomposition, in a *coloring* form (which we will also refer to as *weak network decomposition* or *graph decomposition*), was first introduced in a 1989 paper of Awerbuch et. al. [15]. Since then, there have appeared definitions of a variety of related but different structures, which are all informally placed under the heading "network decomposition", but have also been designated average covers, sparse partitions, the circle problem, sparse neighborhood covers, pairwise covers, etc. Slight variations in definitions of these structures can have large implications for constructibility and applications. As we will see, the weak network decomposition, in many domains, will be the easiest to construct; but it is the sparse neighborhood cover, perhaps the hardest to construct, that seems most useful for the applications. Chapter 1 surveys the different notions and definitions of network decomposition, and provides a standard terminology.

The network decomposition portion of the thesis is structured as follows. A simple greedy construction of weak network decomposition or strong network decomposition is due to Linial-

Saks [47] and Awerbuch-Peleg [21], and we present it for background, along with our survey of alternative definitions for network decomposition in Chapter 1. Linial and Saks [47] also presented a randomized parallel algorithm for weak network decomposition. In Chapter 2, we give a modified randomized parallel *RNC* algorithm for this problem whose analysis can be shown to work with lesser independence, and then remove the randomness, to achieve a deterministic *NC* parallel algorithm.

Chapter 3 moves to the distributed model of computing, where we give the fastest known deterministic algorithms for all alternative notions of network decomposition, including the sparse neighborhood cover definition, with applications. Finally, in Chapter 4, we show how to construct a sparse neighborhood cover in near linear time, using a new "guarded breath first search" technique, and an amortized analysis. This has applications which include fast constructions of planar separators, and approximating shortest paths.

Local resource allocation

In an asynchronous network, one way to perform typical network functions, such as symmetry breaking, or resource allocation, involves globally synchronizing the network and then running a synchronous protocol. Such synchronization, however, can be difficult and expensive. We examine a setting where a local strategy achieves good performance without simulating synchrony.

Chapter 5 gives the first poly-logarithmic time (randomized) protocol for breaking symmetry in a totally asynchronous environment. The protocol can be adapted to the distributed or asynchronous PRAM model of computation. Our means of breaking symmetry is to construct a maximal independent set (MIS) in the following graph: the vertices correspond to the network processors, and two vertices are connected by an edge if there is a direct communication link between the corresponding processors.

We give the first poly-logarithmic time (randomized) protocol for constructing an MIS in a totally asynchronous environment. In addition, the local performance of the algorithm is shown to depend only on local clock delays. (We extend the notion of "wait-free" to graph algorithms, and define k-wait-free as a measure of local dependence).

As an application of the new protocol, a randomized solution to the classical *dining philosophers* problem is presented, that schedules a job j in $O(\delta)$ expected time, where δ is the number

of jobs with which j competes. This meets the lower bound of $\Omega(\delta)$, where the previous best known algorithms ran in time $O(\delta^2)$.

Chapter 1

Preliminaries and Definitions: Decompositions of a Graph into Regions of Low Diameter

This chapter presents the introductory material and notation needed for chapters 2-4. There are several alternate notions of low-diameter graph or network decomposition. These are related, but different. In this section we survey the different formulations of network decomposition, and discuss their relations.

Within each family of definitions, we also discuss what it means to have a *high-quality* decomposition or cover, in terms of the optimal tradeoffs between low diameter and sparsity.

1.1 Introductory definitions

Our definitions consider a graph $G = (V, E)$ whose vertices are grouped into a collection of (possibly overlapping) sets $S_1, \ldots, S_r$ (called also *clusters*). This collection is referred to as a *cover* or *decomposition*.

Our notion of *distance* in a graph is the usual (unweighted) one, i.e., the distance between $u, v \in V$ in G, denoted $dist_G(u, v)$, is the length (in edges) of the shortest path between u and v in G. The distance between two clusters S, S' is analogously defined to be the minimum distance (in G) between any two vertices $v \in S$ and $v' \in S'$. (We extend this to positive edge weights and shortest weight paths in the usual way.)

However, for distances *inside* clusters, we must distinguish between two distinct notions.

Definition 1.1.1 *The* weak *distance between $u, v \in S_i$ is simply their distance in G, $dist_G(u, v)$. Namely, the path connecting them is allowed to shortcut through vertices not in S_i. The* weak diameter *of S_i is defined as*

$$diam(S_i) = \max_{u,v \in S_i}(dist_G(u, v)).$$

Definition 1.1.2 *The* strong *distance between $u, v \in S_i$, denoted $dist_{S_i}(u, v)$, is the length of the shortest path between u and v, on the induced subgraph S_i of G. Namely, all vertices on the path connecting u and v must also be in S_i. The* strong diameter *of S_i is defined as*

$$Diam(S_i) = \max_{u,v \in S_i}(dist_{S_i}(u, v)).$$

The *square* of a graph, G^2, is defined to be the graph G closed to distance 2, i.e., it contains an additional edge (u, v) if there exists an intermediate vertex w s.t. (u, w) and (w, v) are in G. Similarly, G^t is the transitive closure of G to distance t, i.e., it contains an edge between any two vertices that are connected by a path of length t or less in G.

The *j-neighborhood* of a vertex $v \in V$ is defined as $N_j(v) = \{w \mid dist_G(w, v) \leq j\}$. When $j = 1$, we omit the subscript and denote the neighbors of v by $N(v)$. The j-neighborhood of a set of vertices W is defined to be $N_j(W) = \cup_{v \in W} N_j(v)$,

Definition 1.1.3 *Given an undirected graph $G = (V, E)$, a* maximal independent set, *or* MIS *in this graph is a set of vertices M such that*

1. *Every vertex is in M or has a neighbor in M*

2. *If $x \in M$, then none of x's neighbors are in M.*

1.2 Network decomposition

1.2.1 Weak network decomposition

Definition 1.2.1 *For an undirected graph $G = (V, E)$, a (χ, d)-*decomposition *is defined to be a χ-coloring of the nodes of the graph that satisfies the following properties:*

1. *each color class is partitioned into an arbitrary number of disjoint* clusters;

2. *the distance in G between any pair of nodes in a cluster is at most $O(d)$, i.e. the diameter $Diam(G)$ is $O(d)$.*

3. *clusters of the same color are at least distance 2 apart.*

The above definition is equivalent to one first introduced by [15]. It can be thought of as a generalization of the standard graph coloring problems, where χ is the number of colors used, and the clusters are "super-nodes" of diameter $O(d)$.

A natural question to ask is for a given family of graphs, for what values of χ and d does such a decomposition exist. It is immediately clear that there will be a tradeoff between minimizing the number of colors and the diameter. If χ = chromatic number of G colors are allowed, then a ordinary vertex coloring gives diameter 0 clusters; if we allow d = diameter of G, we can color the whole graph with a single color. Awerbuch et. al. [15] when they introduced network decomposition, gave a construction with $\chi, d = O(n^\epsilon)$, for any $\epsilon > 0$. As we will see, this is far from an optimal tradeoff between χ and d. The following simple construction [47, 21] shows that any arbitrary graph on n vertices will have a $(\log n, 2\log n)$-decomposition.

1.2.2 The simple greedy algorithm.

Pick an initial color, say green. We will first construct all green clusters. Pick an arbitrary vertex (call it the *center* vertex), and consider successive balls of radius r around this vertex, for $r = 1, 2, 3 \ldots Diam(G)$. For each r, compare the number of vertices that lie in the interior of the ball (i.e. are also in the ball of radius $r - 1$ around the center vertex), with the number of vertices that are of distance exactly r from the center. It is easy to prove that there always exists an $r \leq \log n$ for which the number of vertices in the interior is greater than the number of vertices on the border. Thus, we take the minimum diameter ball B_r for which this condition holds, color the interior green, and remove the entire ball (including the uncolored border) from the graph. Then we pick another arbitrary vertex, and repeat the whole procedure on $G \setminus B_r$, until all vertices are either colored green, or incident to vertices which are colored green. Then, all uncolored vertices are returned to the graph, and we repeat the above on a new color.

Since the radius r of the clusters in this construction is less than $\log n$, the diameter is at most $2\log n$. Two clusters of the same color will be at least distance 2 apart, because the

borders, which we did not color, act as buffers. Finally, since by construction, every cluster has more vertices in the interior than on its border, at least half the remaining vertices are colored with each new color, and so we color all vertices with at most $\log n$ colors.

Linial and Saks [47] gave a triangulation argument for a family of simplex graphs showing that for $\chi \leq \log n$, d must be $O(\log n)$ (where the constant of the lower bound nearly meets the 2 of the construction above). This generalizes to a range of tradeoffs, as follows.

Definition 1.2.2 *A (χ, d)-decomposition is called* high-quality *if χ is at most $O(dn^{1/d})$.*

Many of the applications that we will see will have a running time dominated by the maximum of χ and d. Therefore, the typical high-quality decomposition which will be of interest, is χ and d both $O(\log n)$.

In the next section, we survey all alternate notion of network decomposition, and give formal definitions for each. We divide the definitions into two classes – the *color* definitions and the *cover* definitions. In Section 1.3, we extend the color definitions to separate clusters by distance greater than 1, and the cover definitions to capture not just the neighbors of a vertex, but the neighbors out to distance λ. We call this λ- *extended* network decomposition.

1.2.3 Strong network decomposition

We have already given the definition of weak diameter network decomposition in the previous section. Now we define strong diameter decomposition.

Definition 1.2.3 *For* strong-diameter *network decomposition, the definition is the same as Definition 1.2.1, except in Property 2, substitute "strong" for "weak" diameter.*

Note that G is a strong diameter network decomposition implies G is a weak-diameter network decomposition, and in fact the simple greedy construction given in the previous section also happens to be a strong-diamter decomposition. As with the weak definition, the "high-quality" tradeoffs are optimal.

We will refer to the two definitions above as the *color* definitions. The next three alternative definitions of network decomposition we will refer to collectively as the *cover* definitions. The cover defintions will all be strong-diameter definitions, but they will require additional structure to capture the neighborhoods of all vertices.

1.2.4 The cover definitions

We first present the definition for sparse neighborhood covers, and then weaken this somewhat to define average covers, and pairwise covers. A sparse neighborhood cover will automatically be both an average cover and a pairwise cover, but the converse does not hold. We consider here a cover which represents the neighbors of every vertex. The extension to neighborhoods to distance λ appears in Section 1.3.

Definition 1.2.4 *A* (χ, d) neighborhood cover *is a collection of sets* $S_1, \ldots, S_r$, *with the following properties:*

1. *For every vertex* v, *there exists a set* S_i *s.t.* $N(v) \subseteq S_i$.
2. $Diam(S_i) \leq O(d)$ *for every set* S_i.
3. *Each vertex belongs to at most* χ *sets.*

A (χ, d)*-neighborhood cover is said to be* sparse, *if* $\chi \leq dn^{1/d}$.

Notice that *sparsity* is a measure of the quality of the cover, analagous to the *high-quality* stricture for the color definitions. Also analogously, there will be a tradeoff betweek diameter of sets in the cover, and the maximum overlap χ of sets at a vertex.

Setting $d = 1$, the set of all balls of radius 1 around each vertex is a sparse neighborhood cover. Setting $d = Diam(G)$, the graph G itself is a sparse neighborhood cover. In the first case, the diameter of a ball is 1, but each vertex could appear in every ball. In the second case, each vertex appears only in a single cluster G, but the diameter of G could be as high as n. Setting $d = \log n$ (the typical and useful setting, for most applications), a sparse $(\log n, O(\log n))$-neighborhood cover is a collection of sets S_i with the following properties: the sets contain all 1-neighborhoods, the diameter of the sets is bounded by $O(\log n)$, and each vertex is contained in at most $c \log n$ sets, where $c > 0$.

A weaker structure that resembles the sparse neighborhood cover, is the *sparse-average-cover* defined first by [21].

Definition 1.2.5 *A* (χ, d) average cover *is a collection of sets* $S_1, \ldots, S_r$, *with the following properties:*

1. *For every vertex* v, *there exists a set* S_i *s.t.* $N(v) \subseteq S_i$.

2. *$Diam(S_i) \leq O(d)$ for every set S_i.*

3. *($\sum_{v \in G}$ number of S_i which contain v)/$v \leq \chi$.*

A (χ, d) average cover is said to be sparse, *if $\chi \leq dn^{1/d}$.*

A still weaker structure, where we again require the cover to be only sparse on average, and cover not the whole neighborhood but only pairs, is the *pairwise cover* of E. Cohen. We remark that the simple greedy algorithm for network decomposition presented above, if we include the borders in the output sets, produces a pairwise cover, but not a sparse pariwise cover. However, modifying the stopping conditions as we do in Chapter 4, will allow us to insure the cover is sparse on average. (This is essentially what E. Cohen does [30]).

Definition 1.2.6 *A (χ, d)* pairwise cover *is a collection of sets $S_1, \ldots, S_r$, with the following properties:*

1. *For every pair of neighboring vertices u, v, there exists a set S_i s.t. u, v appear together in S_i.*

2. *$Diam(S_i) \leq O(d)$ for every set S_i.*

3. *($\sum_{v \in G}$ number of S_i which contain v)/$v \leq \chi$.*

A (χ, d) pairwise cover is said to be sparse, *if $\chi \leq dn^{1/d}$.*

We remark again that a sparse neighborhood cover is automatically also an average cover, and a pairwise cover. We further remark that the color definitions can be thought of as sparse 1-neighborhood covers of the edges (The analogy, however, does not carry over to the λ extensions of Section 1.3).

1.2.5 A composite structure

Finally, we present a structure that rather than weakening the requirements on a sparse neighborhood cover, actually strengthens them. We define a *layered sparse neighborhood cover* which is basically a strong-diameter network decomposition sitting inside a sparse neighborhood cover. This stronger structure is in fact what the sparse neighborhood covers algorithms of Chapters 3 and 4 will construct.

Definition 1.2.7 *A* layered (χ, d)-neighborhood cover *is a collection $\mathcal{X}$ of subcollections of sets, $\mathcal{Y}_1 \ldots \mathcal{Y}_\lambda$ such that:*

1. *$\mathcal{X}$ is a (χ, d)-neighborhood cover.*

2. *If the* interiors *(i.e. thuse nodes in set Y whose 1-neighborhoods are covered by Y) of $\mathcal{Y}_i$ are colored with color i, then the resulting coloring is a strong-diameter (χ, d) network decomposition.*

A layered (χ, d) neighborhood cover is said to be sparse if $\mathcal{X}$ is a sparse neighborhood cover.

We remark that the sparsity condition on the layered neighborhood cover, above, automatically guarantees that the network decomposition inside is high-quality. Notice, however, that the simple greedy algorithm presented above, though it produçes a strong-diameter netowrk decomposition, will not produce a sparse neighborhood cover. The difficulty is that if you throw back in all the borders, you will cover all 1-neighborhoods, but the resulting cover will not be sparse; a node can appear in the borders of many sets in each layer.

1.3 Extension to separation and local neighborhoods

First, we give the weak diameter definition, generalized to allow an additional "separation" parameter, λ. (When $\lambda = 1$, this is the notion of weak network decomposition discussed in the previous section, equivalent to the definitions in [15, 47, 57]).

Definition 1.3.1 *For an undirected graph $G = (V, E)$, a weak (strong)* (χ, d, λ)-decomposition *is defined to be a χ-coloring of the vertices of the graph, i.e., a mapping $\psi : V \mapsto \{1, \ldots, \chi\}$, that satisfies the following properties:*

1. *each color class is partitioned into (an arbitrary number of) disjoint vertex clusters;*

2. *the weak (strong) diameter of any cluster of a single color class is at most $d\lambda$;*

3. *clusters of the same color are at least distance $\lambda + 1$ apart.*

A (χ, d, λ)-*decomposition* is said to be *high-quality* if it achieves the optimal tradeoff; namely, when cluster diameter $= O(d\lambda)$, the coloring number χ is at most $dn^{1/d}$. (Note that the diameter is allowed to grow proportional to λ, but the number of colors stays the same.)

Remark. We can construct a $(\log n, 2\log n, \lambda)$-decomposition by the following modification of the simple greedy procedure in Section 1.2.2. Simply grow the radius of the ball in incremental hops of size λ, rather than of size 1.

Now we give the sparse neighborhood cover definition, extended to λ-neighborhoods. Note that in the quality condition, the diameter increases proportional to λ but the maximum overlap is not allowed to increase with λ.

Definition 1.3.2 *A* (χ, d, λ)-neighborhood cover *is a collection of sets* $S_1, \ldots, S_r$, *with the following properties:*

1. *For every vertex* v, *there exists a set* S_i *s.t.* $N_\lambda(v) \subseteq S_i$.
2. $Diam(S_i) \leq O(d\lambda)$ *for every set* S_i.
3. *Each vertex belongs to at most* χ *sets.*

A (χ, d, λ)*-neighborhood cover is said to be* sparse, *if* $\chi \leq dn^{1/d}$.

The definitions for average covers and pairwise covers can be extended analogously. For a layered sparse neighborhood cover, the interiors of a sparse (χ, d, λ)-neighborhood cover should produce a high-quality (χ, d, λ) network decomposition. Again, a single layer will produce an sparse average cover.

1.4 Summary of results

The following table summarizes what was known about constructing each network decomposition structure in the parallel, distributed and sequential models of computation. Results in **boldface** appear in this thesis.

For simplicity, all results are stated for high-quality colorings, or sparse coverings, with $d = \log n$ and $\lambda = 1$. In the parallel domain, we will be concerned with the PRAM model (see e.g. [31] for precise definitions). In the distributed domain, we will follow Linial's model (see [48], or Section 3.2 of this thesis for an overview and discussion.) We also place, in the first line of this table, the best known results for constructing a *maximal independent set (MIS)*. One reason is because a maximal independent set is, in some sense, a decomposition of a graph into diameter 1 neighborhoods, though this is much less sophisticated than the other forms of

	Sequential	Rand. PRAM	Det. PRAM	Rand Dist.	Det. Dist.
MIS	$O(n)$	$O(\log n)$ [44, 51]	$O(\log n)$ [50]	**$O(\log n)$** [51][a]	$n^{O(1/\sqrt{\log n})}$ [57]
Weak N.D.	$O(E\log^2 n)$	$O(\log^2 n)$ [47]	**$O(\log^5 n)$**[b]	$O(\log^2 n)$ [47]	**$n^{O(1/\sqrt{\log n})}$**
Pairwise Covers	**$O(E\log^2 n)$** [30]	$O(\log^3 n)$[b] [30]		**$O(\log^4 n)$**	**$n^{O(1/\sqrt{\log n})}$**
Average Covers	**$O(E\log^2 n)$**			**$O(\log^4 n)$**	**$n^{O(1/\sqrt{\log n})}$**
Sparse N. Covers	**$O(E\log^3 n)$**			**$O(\log^4 n)$**	**$n^{O(1/\sqrt{\log n})}$**

[a] Chapter 5 of this thesis extends to an asynchronous network

[b] With a $\log n$ blowup in the number of colors

Figure 1-1: Summary of new and known results. (New results are in boldface.)

network decomposition we consider. The second reason is that while a network decomposition can be used to construct an MIS in all models of computation, there are usually simpler ways (either from a complexity-theoretic point of view, or from a conceptual point of view, or both) to construct the MIS directly.

Chapter 2

Low Diameter Graph Decomposition is in NC

2.1 Introduction

In this chapter we achieve the first polylogarithmic-time deterministic parallel algorithm for (χ, d)-decomposition. The algorithm decomposes an arbitrary graph into $O(\log^2 n)$ colors, with cluster diameter at most $O(\log n)$. Thus we place the low-diameter graph decomposition problem into the class *NC*.

The algorithm uses a non-trivial scaling technique to remove the randomness from the algorithm of Linial-Saks. In Section 2.2.1, we review the Linial-Saks algorithm. Section 2.2.2 gives our new modified *RNC* algorithm, whose analysis is shown in Section 2.2.4 to depend only on *pairwise* independence. This is the crux of the argument. Once we have a pairwise independent *RNC* algorithm, it is well known how to remove the randomness to obtain an *NC* algorithm. In Section 2.3.2 we are a bit more careful, however, in order to keep down the blowup in the number of processors. Our (deterministic) *NC* algorithm runs in $O(\log^5(n))$ time and uses $O(n^2)$ processors.

This chapter describes joint work with B. Awerbuch, B. Berger and D. Peleg [24, 12].

2.2 An *RNC* algorithm

In this section, we modifying an RNC algorithm of Linial-Saks to depend only on pairwise independence. and then removing the randomness. To get our newly-devised pairwise independent *benefit function* [44, 50] to work, we have to employ a non-trivial scaling technique. Scaling has been used previously only on the simple measure of node degree in a graph.

2.2.1 The *RNC* algorithm of Linial-Saks

Linial and Saks's randomized algorithm [47] emulates the simple greedy procedure presented in Section 1.2.2. To emulate the greedy algorithm randomly, Linial-Saks still consider each of $O(\log n)$ colors sequentially, but must find a distribution that will allow all center nodes of clusters of the same color to grow out in parallel, while minimizing collisions. If all nodes are allowed to greedily grow out at once, there is no obvious criterion for deciding which nodes should be placed in the color-class in such a way that the resulting coloring is guaranteed both to have small diameter and to contain a substantial fraction of the nodes.

Linial-Saks give a randomized distributed (trivially also an *RNC*) algorithm where nodes compete to be the center node. It is assumed that each node has a unique ID associated with it.[1] In their algorithm, in a given phase they select which nodes will be given color j as follows. Each node flips a candidate radius n-wise independently at random according to a truncated geometric distribution (the radius is never set greater than B, which is set below). Each node y then broadcasts the triple $(r_y, ID_y, d(y,z))$ to all nodes z within distance r_y of y. For the remainder of this paper $d(y,z)$ will denote the distance between y and z in G. (This is sometimes referred to as the *weak* distance, as opposed to the *strong* distance, which is the distance between y and z in the subgraph induced by a cluster which contains them.) Now each node z elects its center node, $C(z)$, to be the node of highest ID whose broadcast it received. If $r_y > d(z,y)$, then z joins the current color class; if $r_y = d(z,y)$, then z remains uncolored until the next phase.

Linial and Saks show that if two neighboring nodes were both given color i, then they both declared the same node y to be their winning center node. This is because their algorithm

[1] As seen below, [illegible] is used for a consistent tie-breaking system: the necessity of assuming unique IDs for tie-breaking depends on when whether one is in the distributed or parallel model of computing. This paper is concerned with parallel computation, so we can freely assume unique IDs in the model.

emulates a greedy algorithm that sequentially processes nodes from highest to lowest ID in a phase. The diameter of the resulting clusters is therefore bounded by $2B$. Setting $B = O(\log n)$, they can expect to color a constant fraction of the remaining nodes at each phase. So their algorithm uses $O(\log n)$ colors. (See their paper [47] for a discussion of trade-offs between diameter and number of colors. Linial-Saks also give a family of graphs for which these trade-offs between χ and d are the best possible.)

The analysis of the above algorithm cannot be shown to work with constant-wise independence; in fact, one can construct graphs for which in a sample space with only constant-wise independence, there will not exist a single good sample point, It even seems doubtful that the Linial-Saks algorithm above would work with polylogarithmic independence. So if we want to remove randomness, we need to alter the randomized algorithm of Linial-Saks.

2.2.2 Overview of the pairwise independent *RNC* algorithm

Surprisingly, we show that there is an alternative *RNC* algorithm where each node still flips a candidate radius and competes to be the center of a cluster, whose analysis can be shown to depend only on pairwise independence.

The new algorithm will proceed with *iterations* inside each *phase*, where a phase corresponds to a single color of Linial-Saks. In each iteration, nodes will grow their radii according to the same distribution as Linial-Saks, except there will be some probability (possibly large) that a node y does not grow a ball at all. If a node decides to grow a ball, it does so according to the same truncated geometric distribution as Linial-Saks, and ties are broken according to unique node ID, as in the Linial-Saks algorithm. We get our *scaled* truncated distribution as follows:

$$\begin{aligned} Pr[r_y = NIL] &= 1-\alpha \\ Pr[r_y = j] &= \alpha p^j(1-p) \quad \text{for } 0 \le j \le B-1 \\ Pr[r_y = B] &= \alpha p^B \end{aligned}$$

The design of the algorithm proceeds as follows: we devise a new *benefit function* whose expectation will be a lower bound on the probability a node is colored by a given iteration (color) of the algorithm, plus pairwise independence will suffice to compute this benefit function. The pairwise-independent benefit function will serve as a good estimate to the n-wise independent "benefit function" of the Linial-Saks algorithm, *whenever nodes y in the graph would not expect to be reached by many candidate radii z.* This is why it is important that some nodes not grow

candidate balls at all.

To maximize the new pairwise-independent benefit function, the probability α that a node grows a ball at all will be scaled according to a measure of local *density* in the graph around it (see the definition of the measure T_y below.) Since dense and sparse regions can appear in the same graph, the scaling factor α, will start small, and double in every iteration of a phase (this is the $O(\log n)$ blowup in the number of colors). We argue that in each iteration, those y's with the density scaled for in that iteration, will have expected benefit lower bounded by a constant fraction. Therefore, in each iteration, we expect to color a constant fraction of these nodes (Lemma 2.2.2). At the beginning of a phase α is reset to reflect the maximum density in the remaining graph that is being worked on. In $O(\log n)$ phases of $O(\log n)$ iterations each, we expect to color the entire graph.

2.2.3 The *RNC* algorithm

Define $T_y = \sum_{z|d(z,y)\leq B} p^{d(z,y)}$, and $\Delta = \max_{\forall y \in G} T_y$. Each phase will have $O(\log n)$ iterations, where each iteration i colors a constant fraction of the nodes y with T_y between $\Delta/2^i$ and $\Delta/2^{i-1}$. *Note that T_y decreases from iteration to iteration, but Δ remains fixed.* Δ is only re-computed at the beginning of a phase. [2]

The algorithm runs for $O(\log n)$ phases of $O(\log n)$ iterations each. At each iteration, we begin a new color. For each iteration i of a phase, set $\alpha = 2^i/(3\Delta)$.

Each node y selects an integer radius r_y pairwise independently at random according to the truncated geometric distribution scaled by α (defined in Section 2.2.2). We can assume every node has a unique ID [47]. Each node y broadcasts (r_y, ID_y) to all nodes that are within distance r_y of it. After collecting all such messages from other nodes, each node y selects the node $C(y)$ of highest ID from among the nodes whose broadcast it received in the first round (including itself), and gets the current color if $d(y, C(y)) < r_{C(y)}$. (A NIL node does not broadcast.) At the end of the iteration, all the nodes colored are removed from the graph.

[2] We remark that the RNC algorithm will need only measure T_y, the density of the graph at y once, in order to determine Δ. In fact any upper bound on max T_y in the graph will suffice, though a sufficiently crude upper bound could increase the running time of the algorithm. The dynamically changing T_y is only used here for the analysis; the randomized algorithm does not need to recalculate the density of the graph as nodes get colored and removed over successive iterations within a phase.

2.2.4 Analysis of the algorithm's performance

We fix a node y and estimate the probability that it is assigned to a color, S. Linial and Saks [47] have lower bounded this probability for their algorithm's phases by summing over all possible winners of y, and essentially calculating the probability that a given winner captures y and no other winners of higher ID capture y. Since the probability that $y \in S$ can be expressed as a union of probabilities, we are able to lower bound this union by the first two terms of the inclusion/exclusion expansion as follows:

$$Pr[y \in S] \geq$$
$$\sum_{z \mid d(z,y) < B} \left(Pr[r_z > d(z,y)] - \sum_{u > z \mid d(u,y) \leq B} Pr[(r_z > d(z,y)) \wedge (r_u \geq d(u,y))] \right)$$

Notice that the above lower bound on the probability that y is colored can be computed using only pairwise independence. This will be the basis of our new benefit function. We will indicate why the Linial and Saks algorithm cannot be shown to work with this weak lower bound.[3] However, we can scale α so that this lower bound suffices for the new algorithm.

More formally, for a given node z, define the following two indicator variables:

$X_{y,z}$: $r_z \geq d(z,y)$

$Z_{y,z}$: $r_z > d(z,y)$

Then we can rewrite our lower bound on $Pr[y \in S]$ as

$$\sum_{z \mid d(z,y) < B} E[Z_{y,z}] - \sum_{\substack{u > z \mid \; d(z,y) < B \\ d(u,y) \leq B}} E[Z_{y,z} X_{y,u}]$$

The *benefit* of a sample point $R = \langle r_1, \ldots, r_n \rangle$ for a single node y, is now defined as

$$B_y(R) = \sum_{z \mid d(z,y) < B} Z_{y,z} - \sum_{\substack{u > z \mid \; d(z,y) < B \\ d(u,y) \leq B}} Z_{y,z} X_{y,u}$$

Hence, our lower bound on $Pr[y \in S]$ is, by linearity of expectation, the expected benefit.

[3] We can, in fact, construct example graphs on which their algorithm will not perform well using only pairwise independence, but in this paper we just point out where the analysis fails.

Recall that $T_y = \sum_{z|d(z,y) \le B} p^{d(z,y)}$. We first prove the following lemma:

Lemma 2.2.1 *If $p \le 1/2$ and $B \ge \log n$ then $E[B_y(R)] \ge (1/2)p\alpha T_y(1-\alpha T_y)$.*

Proof We can rewrite

$$E[B_y(R)] = p\alpha \left(\sum_{z|d(z,y)<B} p^{d(z,y)} \right) - p\alpha^2 \left(\sum_{\substack{u>z| \quad d(z,y)<B \\ d(u,y)\le B}} p^{d(z,y)+d(u,y)} \right)$$

So it is certainly the case that

$$\begin{aligned} E[B_y(R)] &\ge p\alpha \left(\sum_{z|d(z,y)<B} p^{d(z,y)} \right) - p\alpha^2 \left(\sum_{z|d(z,y)<B} p^{d(z,y)} \right) \left(\sum_{u|d(u,y)\le B} p^{d(u,y)} \right) && (2.1) \\ &= p\alpha \left(\sum_{z|d(z,y)<B} p^{d(z,y)} \right) \left(1 - \alpha \left(\sum_{u|d(u,y)\le B} p^{d(u,y)} \right) \right). && (2.2) \\ &= p\alpha \left(\sum_{z|d(z,y)<B} p^{d(z,y)} \right) (1-\alpha T_y). && (2.3) \end{aligned}$$

Now, there are less than n points at distance B from y, and $p \le 1/2$ and $B \ge \log n$ by assumption, so

$$\sum_{z|d(z,y)=B} p^B < np^B \le 1.$$

On the other hand

$$\sum_{z|d(z,y)<B} p^{d(z,y)} \ge 1,$$

since the term where $z = y$ contributes 1 already to the sum. Thus

$$\sum_{z|d(z,y)<B} p^{d(z,y)} \ge \sum_{z|d(z,y)=B} p^B$$

And since there two terms sum to T_y,

$$\sum_{z|d(z,y)<B} p^{d(z,y)} \ge T_y/2.$$

Substituting $T_y/2$ in equation 2.3 yields the lemma. □

Define $\Delta = \max_y(T_y)$. Define the set D_i at the ith iteration of a phase as follows:

$$D_i = \{y | \Delta/2^i \le T_y \le \Delta/2^{i-1} \wedge (y \notin D_h \text{ for all } h < i)\}$$

Given a sample point $R = <r_1, \ldots, r_n>$, define the benefit of the ith iteration of a phase as:

$$BI(R) = \sum_{D_i} B_y(R). \tag{2.4}$$

Recall that $\Delta = \max_{\forall y \in G} T_y$. At the ith iteration of a phase, we will set $\alpha = 2^i/(3\Delta)$. In the analysis that follows, we show that we expect to color a constant fraction of the nodes which have $y \in D_i$ in the ith iteration.

Lemma 2.2.2 *In the ith iteration, we expect to color greater than $p/18$ of those $y \in D_i$.*

Proof

$$\begin{aligned} E[\text{ \# of } y \in S \text{ where } y \in D_i] &= \sum_{y \in D_i} Pr[y \in S] \\ &\ge E[BI(R)] \\ &\ge \frac{p}{2} \sum_{y \in D_i} \left(\frac{2^i}{3\Delta} T_y\right)\left(1 - \frac{2^i}{3\Delta} T_y\right) \end{aligned}$$

by Lemma 2.2.1. Since we want a lower bound, we substitute $T_y \ge \frac{\Delta}{2^i}$ in the positive T_y term and $T_y \le \frac{\Delta}{2^{i-1}}$ in the negative T_y term, giving

$$\begin{aligned} E[\text{ \# of } y \in S \text{ where } y \in D_i] &\ge (p/2) \sum_{y \in D_i} \frac{1}{3}(1 - \frac{2}{3}) \\ &> \frac{p}{18} |D_i|. \end{aligned}$$

□

The next lemma gives the expected number of phases is $O((\log n)/(\log(p/18))) = O(\log n)$.

Lemma 2.2.3 *Suppose $V' \subseteq V$ is the set of nodes present in the graph at the beginning of a phase. After* $\log(3\Delta)$ *iterations of a phase, the expected number of nodes colored is* $(p/18)|V'|$.

Proof Since for all y, $T_y \ge 1$, over all iterations, and since $\alpha \to 1$, then there must exist an iteration where $\alpha T_y \ge 1/10$. Since T_y cannot increase (it can only decrease if we color and

remove nodes in previous iterations), and $\alpha T_y \leq 1/5$ in the first iteration for all y, we know that for each y there exists an iteration in which $1/5 \geq \alpha T_y \geq 1/10$. If i is the first such iteration for a given vertex y, then by definition, $y \in D_i$, and the sets D_i form a partition of all the vertices in the graph. By Lemma 2.2.2, we expect to color $p/18$ of the vertices in D_i at every iteration i, and every vertex is in exactly one set D_i, so we expect to color a $(p/18)$ fraction overall.

□

By Lemma 2.2.3, we have that the probability of a node being colored in a phase is $p/18$. Thus, the probability that there is some node which has not been assigned a color in the first l phases is at most $n(1-(p/18))^l$. By selecting l to be $\frac{18\log n+\omega(1)}{p}$, it is easily verified that this quantity is $o(1)$.

Theorem 2.2.4 *There is a pairwise independent* RNC *algorithm which given a graph* $G = (V, E)$*, finds a* $(\log^2 n, \log n)$*-decomposition in* $O(\log^3 n)$ *time, using a linear number of processors.*

2.3 The *NC* algorithm

2.3.1 The pairwise independent distribution

We have shown that we expect our *RNC* algorithm to color the entire graph with $O(\log^2 n)$ colors, and the analysis depends on pairwise independence. We now show how to construct a pairwise independent sample space which obeys the truncated geometric distribution. We construct a sample space in which the r_i are pairwise independent and where for $i = 1, \ldots, n$:

$$\begin{aligned} Pr[r_i = NIL] &= 1-\alpha \\ Pr[r_i = j] &= \alpha p^j(1-p) \quad \text{for } 0 \leq j \leq B-1 \\ Pr[r_i = B] &= \alpha p^B \end{aligned}$$

Without loss of generality, let p and α be powers of 2. Let $r = B\log(1/p)+\log(1/\alpha)$. Note that since $B = O(\log n)$, we have that $r = O(\log n)$. In order to construct the sample space, we choose $W \in Z_2^l$, where $l = r(\log n+1)$, uniformly at random. Let $W = <\omega^{(1)}, \omega^{(2)}, \ldots, \omega^{(r)}>$, each of $(\log n + 1)$ bits long, and we define $\omega_j^{(i)}$ to be the jth bit of $\omega^{(i)}$.

For $i = 1, \ldots, n$, define random variable $Y_i \in Z_2^r$ such th [illegible] bit is set as

$$Y_{i,k} = <bin(i), 1> \cdot \omega^{(k)},$$

where $bin(i)$ is the $(\log n)$-bit binary expansion of i.

We now use the Y_i's to set the r_i so that they have the desired property. Let t be the most significant bit position in which Y_i contains a 0. Set

$$\begin{aligned} r_i &= NIL && \text{if } t \in [1, .., \log(1/\alpha)] \\ &= j && \text{if } t \in (\log(1/\alpha) + j\log(1/p), .., \log(1/\alpha) + (j+1)\log(1/p)], \text{ for } j \neq B-1 \\ &= B && \text{otherwise.} \end{aligned}$$

It should be clear that the values of the r_i's have the right probability distribution; however, we do need to argue that the r_i's are pairwise independent. It is easy to see [44, 50] that, for all k, the kth bits of all the Y_i's are pairwise independent if $\omega^{(k)}$ is generated randomly; and thus the Y_i's are pairwise independent. As a consequence, the r_i's are pairwise independent as well.

2.3.2 Searching the sample space

We want to search the sample space given in the previous section to remove the randomness from the pairwise independent *RNC* algorithm; i.e. to find a setting of the r_y's in the ith iteration of a phase for which the benefit, $B_{\mathcal{I}}(R)$, is at least as large as the expected benefit, $E[B_{\mathcal{I}}(R)]$.

Since the sample space is generated from r $(\log n)$-bit strings, it thus is of size $2^{r \log n} \leq O(n^{\log n})$, which is clearly too large to search exhaustively. We could however devise a quadratic size sample space which would give us pairwise independent r_y's with the right property (see [44, 49, 7]). Unfortunately, this approach would require $O(n^5)$ processors: the benefit function must be evaluated on $O(n^2)$ different processors simultaneously.

Alternatively, we will use a variant of a method of Luby [50] to binary search a pairwise independent distribution for a good sample point. We can in fact naively apply this method because our benefit function is a sum of terms depending on one or two variables each; i.e.

$$B_{\mathcal{I}}(R) = \sum_{y \in D_i} B_y(R) = \sum_{y \in D_i} \left(\sum_{s | d(s,y) < B} Z_{y,s} - \sum_{\substack{u > s | \; d(s,y) < B \\ d(u,y) \leq B}} Z_{y,s} X_{y,u} \right) \quad (2.5)$$

where recall $D_i = \{y | \Delta/2^i \leq T_y \leq \Delta/2^{i-1} \wedge (y \notin D_h \text{ for all } h < i)\}$. The binary search is over the bits of W (see Section 2.3.1): at the qt-th step of the binary search, $\omega_t^{(q)}$ is set to 0 if $E[B_{\mathcal{I}}(R) \mid \omega_1^{(1)} = b_{11}, \omega_2^{(1)} = b_{12}, \ldots, \omega_t^{(q)} = b_{qt}]$, with $b_{qt} = 0$ is greater than with $b_{qt} = 1$; and 1 otherwise. [4] The naive approach would yield an $O(n^3)$ processor NC algorithm, since we require one processor for each term of the benefit function, expanded as a sum of functions depending on one or two variables each.

The reason the benefit function has too many terms is that it includes sums over pairs of random variables. Luby gets around this problem by computing conditional expectations on terms of the form $\sum_{i,j \in S} X_i X_j$ directly, using $O(|S|)$ processors. We are able to put our benefit function into a form where we can apply a similar trick. (In our case, we will also have to deal with a "weighted" version, but Luby's trick easily extends to this case.)

The crucial observation is that, by definition of $Z_{y,z}$ and $X_{y,z}$, we can equivalently write $E[Z_{y,z} X_{y,u}]$ as $pE[X_{y,z} X_{y,u}]$; thus, we can lower bound the expected performance of the algorithm within at least a multiplicative factor of p of its performance in Lemmas 2.2.2 and 2.2.3, if we upper bound the latter expectation.

It will be essential throughout the discussion below to be familiar with the notation used for the distribution in Section 2.3.1. Notice that our indicator variables have the following meaning:

$$
\begin{aligned}
X_{y,z} &\equiv Y_{z,k} = 1 \quad \text{for all } k,\ 1 \leq k \leq d(z,y)\log(1/p) \\
Z_{y,z} &\equiv Y_{z,k} = 1 \quad \text{for all } k,\ 1 \leq k \leq (d(z,y)+1)\log(1/p)
\end{aligned}
$$

If we fix the outer summation of the expected benefit at some y, then the problem now remaining is to show how to compute

$$
E[\sum_{(z,u) \in S} X_{y,z} X_{y,u} \mid \omega_1^{(1)} = b_{11}, \omega_2^{(1)} = b_{12}, \ldots, \omega_t^{(q)} = b_{qt}], \tag{2.6}
$$

in $O(\log n)$ time using $O(|S|)$ processors. For notational convenience, we write (z,u) for $z \neq u$. Below, we assume all expectations are conditioned on $\omega_1^{(1)} = b_{11}, \ldots, \omega_t^{(q)} = b_{qt}$.

[4] We remark that to evaluate the benefit of a sample point, we must be able to determine for a given iteration i of a phase, which y are in D_i. Thus we must update T_y for each y to reflect the density of the remaining graph at iteration i.

Note that we only need be interested in the case where both random variables $X_{y,z}$ and $X_{y,u}$ are undetermined. If $q > d(i,y)\log(1/p)$, then $X_{y,i}$ is determined. So we assume $q \leq d(i,y)\log(1/p)$ for $i = z, u$. Also, note that we know the exact value of the first $q-1$ bits of each Y_z. Thus, we need only consider those indices $z \in S$ in Equation 2.6 with $Y_{z,j} = 1$ for all $j \leq q-1$; otherwise, the terms zero out. Let $S' \subseteq S$ be this set of indices.

In addition, the remaining bits of each Y_z are independently set. Consequently,

$$\begin{aligned} E[\sum_{(z,u)\in S'} X_{y,z}X_{y,u}] &= E[\sum_{(z,u)\in S'} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q}] \\ &= E[(\sum_{z\in S'} \gamma(z,y)Y_{z,q})^2 - \sum_{z\in S'} \gamma(z,y)^2 Y_{z,q}^2], \end{aligned}$$

where $\gamma(z,y) = 1/2^{d(z,y)\log(1/p)-q}$

Observe that we have set t bits of $\omega^{(q)}$. If $t = \log n + 1$, then we know all the $Y_{z,q}$'s, and we can directly compute the last expectation in the equation above. Otherwise, we partition S' into sets $S_\lambda = \{z \in S' \mid z_{t+1}\cdots z_{\log n} = \lambda\}$. We further partition each S_λ into $S_{\lambda,0} = \{z \in S_\lambda \mid \sum_{i=1}^{t} z_i\omega_i^{(q)} = 0 \pmod 2\}$ and $S_{\lambda,1} = S_\lambda - S_{\lambda,0}$. Note that given $\omega_1^{(1)} = b_{11}, \ldots, \omega_t^{(q)} = b_{qt}$,

1. $Pr[Y_{z,q} = 0] = Pr[Y_{z,q} = 1] = 1/2$,

2. if $z \in S_{\lambda,j}$, and $u \in S_{\lambda,j'}$, then $Y_{z,q} = Y_{u,q}$ iff $j = j'$, and

3. if $z \in S_\lambda$ and $z' \in S_{\lambda'}$, where $\lambda \neq \lambda'$, then $Pr[Y_{z,q} = Y_{u,q}] = Pr[Y_{z,q} \neq Y_{u,q}] = 1/2$.

Therefore, conditioned on $\omega_1^{(1)} = b_{11}, \ldots, \omega_t^{(q)} = b_{qt}$,

$$\begin{aligned} &E[\sum_{(z,u)\in S'} X_{y,z}X_{y,u}] \\ &= E[\sum_{(z,u)\in S'} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q}] \\ &= E[\sum_{\lambda}\sum_{(z,u)\in S_\lambda} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q} + \sum_{(\lambda,\lambda')}\sum_{z\in S_\lambda}\sum_{u\in S_{\lambda'}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q}] \\ &= \sum_{\lambda} E[\sum_{(z,u)\in S_{\lambda,0}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q} + \sum_{(z,u)\in S_{\lambda,1}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q} \\ &\quad + 2\sum_{z\in S_{\lambda,0}}\sum_{u\in S_{\lambda,1}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q}] + \sum_{(\lambda,\lambda')} E[\sum_{z\in S_\lambda}\sum_{u\in S_{\lambda'}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q}] \\ &= \sum_{\lambda}\left[\frac{1}{2}\sum_{(z,u)\in S_{\lambda,0}} \gamma(z,y)\gamma(u,y) + \frac{1}{2}\sum_{(z,u)\in S_{\lambda,1}} \gamma(z,y)\gamma(u,y) + 0\right] \end{aligned}$$

$$+\sum_{(\lambda,\lambda')}\frac{1}{4}\left(\sum_{z\in S_\lambda}\gamma(z,y)\right)\left(\sum_{u\in S_{\lambda'}}\gamma(u,y)\right)$$

$$=\frac{1}{2}\sum_{\lambda}\left[\left(\sum_{z\in S_{\lambda,0}}\gamma(z,y)\right)^2-\sum_{z\in S_{\lambda,0}}\gamma(z,y)^2+\left(\sum_{z\in S_{\lambda,1}}\gamma(z,y)\right)^2-\sum_{z\in S_{\lambda,1}}\gamma(z,y)^2\right]$$

$$+\frac{1}{4}\left[\left(\sum_{\lambda}\sum_{z\in S_\lambda}\gamma(z,y)\right)^2-\sum_{\lambda}\left(\sum_{z\in S_\lambda}\gamma(z,y)\right)^2\right]$$

Since every node $z \in S'$ is in precisely four sums, we can compute this using $O(|S|)$ processors.

In the above analysis, we fixed the outer sum of the expected benefit at some y. To compute the benefit at iteration i, we need to sum the benefits of all $y \in D_i$. However, we argued in the proof of Lemma 2.2.3 that the sets D_i form a partition of the vertices. Therefore we consider each y exactly once over all iterations of a phase, and so our algorithm needs only $O(n^2)$ processors, and we obtain the following theorem.

Theorem 2.3.1 *There is an* NC *algorithm which given a graph* $G = (V, E)$, *finds a* $(\log^2 n, \log n)$*-decomposition in* $O(\log^5 n)$ *time, using* $O(n^2)$ *processors.*

Chapter 3

Fast Deterministic Distributed Network Decomposition

3.1 Introduction

This chapter presents a deterministic sublinear-time distributed algorithm for all alternate definitions of high-quality network decomposition appearing in Chapter 1. Most importantly for network applications, this chapter yields a fast (logarithmic time randomized, or sub-linear time deterministic) distributed algorithm for constructing a diameter $O(\log n)$, maximum overlap $O(\log n)$ *sparse neighborhood 1-cover* of a network. This improves the distributed pre-processing time for all-pairs shortest paths, load balancing, broadcast, and bandwidth management.

The reader is referred to Chapter 1 for the definitions and notation we will use throughout this chapter.

3.1.1 Previous work

Previous work on the distributed construction of neighborhood covers and decompositions can be classified into three main groups.

Simple greedy approaches: Previous to this work (which also appears in [11]), there were no deterministic constructions of *high-quality* decompositions previously known for distributed

This chapter describes joint work with B. Awerbuch, B. Berger and D. Peleg [24, 11].

networks, that were any different than just applying sequential greedy algorithms. Even for weak or strong network decomposition (coloring definitions) a distributed implementation of the inherently sequential simple greedy algorithm of Section 1.2.2 took $O(n \log n)$ time.

Recursive coloring: There were, however, distributed near-linear constructions of low-quality network decompositions (coloring definitions). First, a fast algorithm is given in [15] for obtaining a (weak-diameter) network decomposition with $O(n^{\epsilon})$-diameter clusters, for $\epsilon = O(\sqrt{\log\log n}/\sqrt{\log n})$. This algorithm requires $O(n^{\epsilon})$ time in the distributed case, and $O(nE)$ sequential operations. Later, the distributed running time for this task was reduced in [57] to $O(n^{\bar{\epsilon}})$, where $\bar{\epsilon} = O(1/\sqrt{\log n})$. (This was independently, and at the same time as the work described in [11] first appeared.)

Unfortunately, the constructions of [15, 57] are not high-quality: they are inefficient in terms of the tradeoff between diameter and number of colors. Roughly speaking, the inefficiency factor is $O(n^{\epsilon})$, and this factor carries over to all but some of the graph-theoretic applications, rendering the decompositions of [15, 57] expensive in a number of practical contexts. These constructions are, nevertheless, sufficient for the two main applications that [15, 57] are primarily concerned with, namely, the maximal independent set problem and $(\Delta + 1)$ coloring a graph of maximum degree Δ. This is because to construct an MIS or a $(\Delta + 1)$ coloring, one needs to traverse the $O(n^{\epsilon})$-diameter clusters only a constant number of times. In contrast, network control applications, such as routing, online tracking of mobile users, and all-pairs shortest paths, require us to traverse the clusters many times. Therefore, a higher-quality decomposition is needed to avoid a large blowup in the running time for these latter applications. (The network applications also need the extensions to λ neighborhoods, which we achieve with the standard extra factor of λ blowup.)

A randomized algorithm The randomized algorithm of [47] achieves a high-quality weak network decomposition by introducing randomization. In this chapter, we can extend this to all notions of network decomposition.

In summary, all three previous approaches fell short of achieving the goal of constructing a *high-quality* network decomposition (color *and* cover definitions) *deterministically* and in *sublinear time*; or a randomized *strong color* or *1-neighborhood cover* high-quality decomposition in expected $O(\chi d)$ time.

3.1.2 Structure of the chapter:

We first review the distributed model of Linial [48], in which we measure the performance of our algorithms. Linial's model is like the PRAM model, in that it is theoretically very elegant, but perhaps sweeps "under the rug" many issues which affect the performance of practical implementations. The model does, however, seem to provide an excellent measure of how well we have localized information in our algorithms. We discuss these issues further along with the model in the next section.

Having established the model, in Section 3.3 we present our recursive algorithm for weak network decomposition which constructs a high-quality network decomposition, in sublinear time. In Section 3.4 we present the reduction which efficiently transforms a weak network decomposition to any of the alternative notions of network decomposition. Coupled with the algorithm in Section 3.3 or [57], this gives a sublinear time deterministic distributed algorithm for all alternative notions of network decomposition. Coupled with the randomized algorithm of Linial-Saks [47] (also see a sketch of the Linial-Saks algorithm in Section 2.2.1 of this thesis), this gives an polylogarithmic randomized algorithm for all alternative notions of network decomposition.

We also give the standard [21] extension for neighborhoods to distance λ, (with the standard factor of λ blowup), which is needed for the applications.

3.2 Linial's model

Linial's model (see [48]) of distributed computing is quite powerful. There is a graph $G = (V, E)$ each node of which is occupied by a processor. Computation is completely synchronous, and reliable. Every time unit each processor may pass messages to each of his neighbors. There is no limit on size of these messages. Also, we do not charge for the time it takes individual processors to compute functions; we only require that these are polynomial-time computations.

We remark that in Linial's model, in $O(d)$ time, where d is the diameter of the network, we can compute any polynomial function of the graph G. This is because in $O(d)$ time a single node can learn the entire network topology, perform all computation at its associated processor (where we do not charge for work by a single processor), and then broadcast the result to all the nodes in the graph.

The interest in Linial's model is that the $\Omega(d)$ threshold presents a mathematically clean measure of the "globalness" of computation. What functions of G can be computed in less than $\Omega(d)$ time? Any sub-diametric algorithms are computing global functions without global knowledge of graph topology.

Much previous work has used network decomposition as a data structure to speed up distributed algorithms using its representation of local neighborhoods. It is thus good to find we can construct such representations locally, in sub-diametric time in Linial's model.

We remark that more realistic models of distributed systems (if we are more interested in practical performance, rather than a mathematically clean concept of the complexity of global communication) limit the size of messages that can be sent in one time unit over an edge, charge for local space and time, can remove the assumption that the system is synchronous, and can also seek to handle various faults or a dynamically changing network. The results presented in Chapter 5 of this thesis present symmetry breaking results in such a realistic model. For ideas on how one might go about adapting the algorithms in *this* chapter to a dynamic asynchronous environment, the reader is referred to the transformer techniques in [2, 21, 19].

On the other hand, in a static asynchronous environment, if the algorithm that runs on top of a network decomposition data structure requires messages to traverse the diameter of the network, then for a low number of messages, we suggest instead constructing the network decomposition using the "sequential" algorithm of Chapter 4.

3.3 Weak diameter network decomposition

In this section, we introduce the new distributed algorithm `Color`, which recursively builds up a $(dn^{1/d}, 2d, 1)$-decomposition. It calls on a procedure, `Create_New_Color`, which runs a modified version of the Awerbuch-Peleg [9] greedy algorithm on separate clusters.

Note that all distances in the discussion below, including those in the same cluster, are assumed to be *weak* distances, and the diameter of the clusters is always in terms of weak diameter (see Definition 1.1.1).

3.3.1 Algorithm `Color`

`Color` is implicitly taking higher and higher powers of the graph, where recall that we define

the graph G^t to be the graph in which an edge is added between any pair of nodes that have a path of length $\leq t$ in G. Notice that to implement the graph G^t in a distributed network G, since the only edges in the network are still the edges in the underlying graph G, to look at all our neighbors in the graph G^t, we might have to traverse paths of length t. Therefore the time for running an algorithm on the graph G^t, blows up by a factor of t. The crucial observation is that a $(\chi, d, 1)$-decomposition on G^t is a (χ, dt, t)-decomposition on G. Choosing t well at the top level of the recursion, guarantees that nodes in different clusters of the same color are always separated by at least twice the maximum possible distance of their radii. We can thus use procedure `Greedy_Color` to in parallel *recolor* these separate clusters without collisions. (The leader of each cluster does all the computation for its cluster.)

The recursive algorithm has two parts:

1. Find a (χ, dt, t)-decomposition, where $\chi = xkn^{1/k}$, $d, t = 2k$, on each of x disjoint subgraphs.

2. Merge these together by recoloring, as just described, to get a $(kn^{1/k}, 2k, 1)$-decomposition.

Algorithm: `Color`(G)

Input: graph $G = (V, E)$, $|V| = n$, and integer $k \geq 1$.
Output: A $(kn^{1/k}, 2k, 1)$-decomposition of G.

1. Compute G^{2k}.

2. If G has less than x nodes, run the Linial-Saks [47] or Awerbuch-Peleg [9] simple greedy algorithm on G^{2k}, and go to step 6.

3. Partition nodes of G into x subsets, $V_1, \ldots, V_x$ (based on the last $\log x$ bits of node IDs, which are then discarded).

4. Define G_i to be the subgraph of G^{2k} induced on V_i.

5. In parallel, for i, `Color`(G_i).
   (every node of G is now colored recursively)

6. For each $v \in V$, color v with the color $<i, \text{color}(v) \in G_i>$.
   (this gives an $xkn^{1/k}$ coloring of G with separation 2k)

7. Do sequentially, for $i = 1$ to $kn^{1/k}$, `Create_New_Color`(G, i)
   (this gives a $kn^{1/k}$ coloring of G with separation 1)

Algorithm: `Create_New_Color`(G, i)
(this colors a constant fraction of the old-colored nodes remaining with new color i)

Input: graph G with new and old colored nodes such that there is a $(xkn^{1/k}, (2k)^2, 2k)$-decomposition on the old-colored nodes of G and a $(i-1, 2k, 1)$-decomposition on the new-colored nodes of G

Output: graph G with new and old colored nodes such that there is a $(xkn^{1/k}, (2k)^2, 2k)$-decomposition on the old-colored nodes of G and a $(i-1, 2k, 1)$-decomposition on the new-colored nodes of G

1. $W \leftarrow V$.

2. Do sequentially, for $j = 1$ to $xkn^{1/k}$,
   "Look at nodes with old color j":

   (a) Do in parallel for color j clusters,

   - Elect a leader for each cluster.
   - The leader learns the identities, U, of all the nodes in W within k distance from the border of its cluster (i.e. this is graph G for that cluster).
   - The leader calls procedure `Greedy_Color`(R, U), where R is the set of old-colored j nodes in both the leader's cluster and in W.
   - `Greedy_Color` returns (DR, DU). The leader colors the nodes in DR with new color i, and sets $W \leftarrow W - DU$.

`Greedy_Color` is the procedure of the Awerbuch-Peleg [9] greedy algorithm that determines what nodes will be given the current new color. The algorithm identifies a $n^{1/k}$ fraction of the nodes in the cluster R to be colored. The algorithm picks an arbitrary node in R (call it a *center* node) and greedily grows a ball around it of minimum radius r, such that a $n^{1/k}$ fraction of the nodes in the ball lie in the interior (i.e. are in the ball of radius $r-1$ around the center node). It is easy to prove that there always exists an $r \leq k|R|^{1/k}$ for which this condition holds. Note that although the centers of the balls grown out are always picked (arbitrarily) from the

nodes in R, the interiors and borders of the balls which are then claimed, include any of the nodes in U (not just those in R) within the ball. Then another arbitrary node is picked, and the same thing is done, until all nodes in R have been processed. Procedure `Create_New_Color` will then color the interiors of the balls (set DR) with new color i, and remove each entire ball from the working graph W.

Algorithm: `Greedy_Color`(R, U)

Input: sets of nodes R and U, where R is the set of nodes in the cluster and U is a superset of nodes that contains R.

Output: (DR, DU). This returns a constant fraction of the nodes in R in set DR and the 1-neighborhoods of the clusters of DR in set DU.

1. $DR \leftarrow \emptyset$; $DU \leftarrow \emptyset$.
2. While $R \neq \emptyset$ do
   (a) $S \leftarrow \{v\}$ for some $v \in R$.
   (b) While $|N_1(S) \cap U| > |R|^{1/k}|S|$ do $S \leftarrow S \cup (N_1(S) \cap U)$.
   (c) $DR \leftarrow DR \cup S$.
   (d) $DU \leftarrow DU \cup (N_1(S) \cap U)$.
   (e) $R \leftarrow R - S - (N_1(S) \cap R)$.
   (f) $U \leftarrow U - S$.

3.3.2 Analysis of `Color`

Lemma 3.3.1 *If* $x = 2^{\sqrt{\log n}\sqrt{1+\log k}}$, *the running time of the procedure* `Color` *is* $n^{2\sqrt{1+\log k}/\sqrt{\log n}+2/k}(2k)^2$.

Proof The branching phase of the recursion takes time $T'(n) \leq 2kT'(n/x) + x$. The merge takes time $\chi k n^{1/k} td = x(kn^{1/k})^2(2k)^2$, where $\chi k n^{1/k}$ is the number of iterations overall and td is the number of steps per iteration. Overall, we have

$$T(n) \leq 2kT(n/x) + x(kn^{1/k})^2(2k)^2$$

$$\leq (2k)^{\log n/\log x} x(kn^{1/k})^2(2k)^2$$
$$\leq n^{2\sqrt{1+\log k}/\sqrt{\log n}+2/k}(2k)^2,$$

when $x = 2^{\sqrt{\log n}\sqrt{1+\log k}}$. □

Theorem 3.3.2 *There is a deterministic distributed asynchronous algorithm which given a graph $G = (V, E)$, finds a $(kn^{1/k}, 2k, 1)$-decomposition of G in $n^{2\sqrt{1+\log k}/\sqrt{\log n}+2/k}(2k)^2$ time.*

Corollary 3.3.3 *There is a deterministic distributed asynchronous algorithm which given $G = (V, E)$, finds a $(O(\log n), O(\log n), 1)$-decomposition of G in $n^{O(\sqrt{\log\log n}/\sqrt{\log n})}$ time, which is n^ϵ for any $\epsilon > 0$. We remark that the constant on the big-oh in the running time is 3.*

Independently, and at the same time as we introduced the above algorithm in [11], [57] obtained a slightly better asymptotic running time for low-quality weak-diameter network decomposition than Awerbuch et. al. [15]. In the next section, we show how to use the result of [57] coupled with our *transformer* algorithm to obtain the same running time for a deterministic construction of *high-quality* weak network decomposition. We thus obtain the following corollary:

Corollary 3.3.4 *There is a deterministic distributed asynchronous algorithm which given $G = (V, E)$, finds a $(O(\log n), O(\log n), 1)$-decomposition of G in $O(n^{1/\sqrt{\log n}})$ time, which is n^ϵ for any $\epsilon > 0$.*

3.4 The transformer algorithm

We introduce an algorithm `Transform`, which takes as input parameters d and λ, and a procedure `Decomp`, which given a graph $G = (V, E)$, finds a $(\chi_{old}, d_{old}, 1)$-decomposition of G. In actuality, we will bind `Decomp` to the algorithm of [57] if we want a deterministic algorithm, and to the algorithm of Linial-Saks [47] if we want a randomized algorithm. `Transform` first calls procedure `Decomp` with $G^{4d\lambda}$. Of course, this will yield an $O(d\lambda)$ blowup in the running time of `Decomp`, say τ. Then `Transform` will call either a modification of the simple greedy algorithm, if we want a high-quality (weak or strong diameter) network decomposition (color definitions), or a modification of Awerbuch-Peleg's sequential cover algorithm [21] if we want a sparse neighborhood cover, in parallel over separate clusters.

Once Decomp is called, the remaining running time for Transform is $O(d_{old}\chi_{old}d^2n^{1/d})$, times a λ blowup for traversing λ-neighborhoods. Then, in sum, Transform is able to obtain a sparse λ-neighborhood cover, or a high-quality (χ, d, λ)-network decomposition in the original graph G in time $O(\lambda d\tau + \lambda d_{old}d\chi_{old}dn^{1/d})$. Recall that, for the applications, we typically set $d = \log n$.

Notation. In the algorithms below, we use roman capital letters for names of sets, and calligraphic letters for names of collections of sets. In particular, corresponding to a set W, by convention we will denote by $\mathcal{W}$ the collection consisting of the sets $\{N_\lambda(v)|v \in W\}$.

3.4.1 Algorithm Transform

Algorithm: Transform(G,Decomp,Re_Cover)

Input: A graph $G = (V, E)$, $|V| = n$, and integer $d \geq 1$, a procedure Decomp, that finds a $(\chi_{old}, d_{old}, 1)$-decomposition of G.
Output: If Re_Cover is bound to a sequential high-quality coloring algorithm, T, a (χ, d, λ) high-quality network decomposition. If Re_Cover is bound to a sequential sparse covering algorithm, $\mathcal{T}$, a sparse (χ, d, λ)-neighborhood cover of G.

1. Decomp($G^{4d\lambda}$).
   (returns a $(\chi_{old}, d_{old}, 1)$-decomposition of $G^{4d\lambda}$ which is a $(\chi_{old}, 4d\lambda d_{old}, 4d\lambda)$-decomposition of G.)
   (a) $\mathcal{T} \leftarrow \emptyset$.
      ($\mathcal{T}$ is the decomp. or cover.)
   (b) Do sequentially, for $i = 1$ to $dn^{1/d}$,
      (find a $dn^{1/d}$-degree coloring or cover of G.)
      i. $\mathcal{U} \leftarrow \{N_\lambda(v)|v \in V\}$.
         ($\mathcal{U}$ is the collection of all unprocessed neighborhoods, or un new-colored nodes.)
      ii. Do sequentially, for $j = 1$ to χ_{old},
         "Look at vertices with old color j":
         A. Do in parallel for color j clusters,
            - Elect a leader for each cluster.

- The leader learns the identities, of all the λ-neighborhoods of vertices within a $2d\lambda$ distance from the border of its cluster.
- The leader calls procedure Re_Cover($\mathcal{R},\mathcal{U}$) on G, where $\mathcal{R}$ is the collection of λ-neighborhoods of old-colored j vertices in both the leader's cluster and in U.
- Re_Cover returns $(\mathcal{DR},\mathcal{DU})$. The leader colors the vertices in $\mathcal{DR}$ with new color i, and sets $\mathcal{U} \leftarrow \mathcal{U} - \mathcal{DU}$.

iii. $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{DR}$

Re_Cover is bound either to procedure Greedy_Color of the previous section, or procedure Cover, which is now presented below.

Cover is our modification of the Awerbuch-Peleg [21] coarsening algorithm that determines what vertices will be given the current new color. The actual code for this procedure follows a description of the algorithm below. The key to our fast simulation of their coarsening algorithm, is that we keep track of neighborhoods within and outside of the old-colored j clusters separately, in order to recolor clusters in parallel without collisions.

3.4.2 The modified procedure Cover

Procedure Cover($\mathcal{R},\mathcal{U}$) operates in iterations. Each iteration constructs one output cluster $Y \in \mathcal{DT}$, by merging together some clusters of $\mathcal{U}$. The iteration begins by arbitrarily picking a cluster S in $\mathcal{U} \cap \mathcal{R}$ and designating it as the kernel of a cluster to be constructed next. The cluster is then repeatedly merged with intersecting clusters from $\mathcal{U}$. This is done in a layered fashion, adding one layer at a time. At each stage, the original cluster is viewed as the internal kernel Y of the resulting cluster Z. The merging process is carried repeatedly until reaching a certain sparsity condition (specifically, until the next iteration increases the number of clusters merged into Z by a factor of less than $|\mathcal{R}|^{1/d}$). The procedure then adds the kernel Y of the resulting cluster Z to a collection $\mathcal{DT}$. It is important to note that the newly formed cluster consists of only the kernel Y, and not the entire cluster Z, which contains an additional "external layer" of $\mathcal{R}$ clusters. The role of this external layer is to act as a "protective barrier" shielding the generated cluster Y, and providing the desired disjointness between the different clusters Y added to $\mathcal{DT}$.

```
DT ← ∅; DR ← ∅
repeat
    Select an arbitrary cluster S ∈ U ∩ R.
    Z ← {S}
    repeat
        Y ← Z
        Y ← ∪_{S∈Y} S
        Z ← {S | S ∈ U, S ∩ Y ≠ ∅}.
    until |Z| ≤ |R|^{1/d} |Y|
    U ← U − Z
    DT ← DT ∪ {Y}
    DR ← DR ∪ Y
until U ∩ R = ∅
Output (DR, DT).
```

Figure 3-1: Procedure $\texttt{Cover}(\mathcal{R},\mathcal{U})$.

Throughout the process, the procedure keeps also the "unmerged" collections $\mathcal{Y}, \mathcal{Z}$ containing the original $\mathcal{R}$ clusters merged into Y and Z. At the end of the iterative process, when Y is completed, every cluster in the collection $\mathcal{Y}$ is added to $\mathcal{DR}$, and every cluster in the collection $\mathcal{Z}$ is removed from $\mathcal{U}$. Then a new iteration is started. These iterations proceed until $\mathcal{U} \cap \mathcal{R}$ is exhausted. The procedure then outputs the sets $\mathcal{DR}$ and $\mathcal{DT}$.

Procedure Cover is formally described in Figure 3-1. Its properties are summarized by the following lemma. We comment that our modifications do not change the lemma.

Lemma 3.4.1 **([21])** *Given a graph $G = (V, E)$, $|V| = n$, a collection of clusters $\mathcal{R}$ and an integer d, the collections $\mathcal{DT}$ and $\mathcal{DR}$ constructed by Procedure* Cover$(\mathcal{R},\mathcal{U})$ *operates in iterations. satisfy the following properties:*

(1) All clusters in $\mathcal{DR}$ have their λ-neighborhood contained in some cluster in $\mathcal{DT}$.

(2) $Y \cap Y' = \emptyset$ for every $Y, Y' \in \mathcal{DT}$,

(3) $|\mathcal{DR}| \geq |\mathcal{R}|^{1-1/d}$, and

(4) $\max_{T\in\mathcal{DT}} Diam(T)$

$\leq (2d-1) \max_{R\in\mathcal{R}} Diam(R)$.

3.4.3 The resulting algorithms

Calling algorithm **Transform** with **Decomp** bound to the network decomposition algorithm of [57] gives the following theorem:

Theorem 3.4.2 *There is a deterministic distributed algorithm that given a graph $G = (V, E)$, $|V| = n$, and integers $d, \lambda \geq 1$, constructs a $(dn^{1/d}, d, \lambda)$-neighborhood cover of G in $\lambda n^{O(1/\sqrt{\log n})}$ time, where each vertex is in at most $\chi = O(dn^{1/d})$ clusters, and the maximum strong cluster diameter is $Diam(S_i) = O(d\lambda)$.*

Calling algorithm **Transform** with **Decomp** bound to the randomized network decomposition algorithm of [47] gives the following theorem:

Theorem 3.4.3 *There is a randomized distributed algorithm that given a graph $G = (V, E)$, $|V| = n$, and integers $d, \lambda \geq 1$, constructs a $(dn^{1/d}, d, \lambda)$-neighborhood cover of G in $\lambda O(d^2 \log^2 n^{1/d})$ time, where each vertex is in at most $\chi = O(dn^{1/d})$ clusters, and the maximum strong cluster diameter is $Diam(S_i) = O(d\lambda)$.*

Corollary 3.4.4 *There is a randomized distributed algorithm that given a graph $G = (V, E)$, $|V| = n$, constructs a $(\log n, \log n, 1)$-neighborhood cover of G in $O(\log^4 n)$ time, and a $(\log n, \lambda \log n, \lambda)$-neighborhood cover of G in $O(\lambda \log^4 n)$ time.*

We remark that in place of **Cover**, one could use the more efficient near-linear construction of sparse neighborhood covers presented in the next Chapter (Chapter 4). However, in Linial's model, this does not change the distributed complexity of the algorithms in this Chapter.

Chapter 4

A Near Linear Construction of Sparse Neighborhood Covers with Applications

4.1 Introduction

In this chapter we speed up the construction of sparse neighborhood covers from time $O(nE)$ to $O((Ed + nd^2)n^{2/d} \log n)$, which is $O(E \log^2 n + n \log^3 n)$ when we set $d = \log n$. The definition of a sparse neighborhood cover has already appeared in Section 1.2.4, with discussion, but we repeat it here to make the chapter self-contained. Recall that

Definition 4.1.1 *A* (χ, d, λ)-neighborhood cover *is a collection of sets* $S_1, \ldots, S_r$, *with the following properties:*

1. *For every vertex* v, *there exists a set* S_i *s.t.* $N_\lambda(v) \subseteq S_i$.
2. $Diam(S_i) \leq O(d\lambda)$ *for every set* S_i.
3. *Each vertex belongs to at most* χ *sets.*

A (χ, d, λ)*-neighborhood cover is said to be* sparse, *if* $\chi \leq dn^{1/d}$.

This chapter describes joint work with B. Awerbuch, B. Berger and D. Peleg [13].

Since χ is defined in terms of d and λ in the case of a *sparse* neighborhood cover, we will sometimes abbreviate a $(dn^{1/d}, d, \lambda)$-sparse neighborhood cover as a (d, λ)-sparse neighborhood cover.

The previous algorithms for constructing sparse neighborhood covers took time $O(nE)$ [21] because they required the construction of BFS trees to distance λ from *all* the vertices in the graph. We show how to construct the cover only doing BFS's from carefully selected vertices. Using new techniques, the area of the BFS trees are carved out in such a way that one can bound the overlap with other BFS trees. We shall see that bounding the overlap between BFS trees poses significant difficulties, requiring a complex growth process and an amortized analysis.

Our fast near-linear step algorithm makes sparse neighborhood covers a viable *data structure* for sequential algorithms, and we obtain near-linear approximation algorithms for a variety of fundamental graph problems. For the applications listed below, one first constructs a sparse neighborhood cover in $\tilde{O}(E)$ time, [1] and then produces, for example, an approximate shortest path between any two vertices in time $O(\log^2 n) = \tilde{O}(1)$ time. We also remark that in fact, this chapter constructs a layered sparse neighborhood cover (see Definition 1.2.7).

Application: approximate shortest paths. The *k-pairs shortest paths* problem is: given a graph $G(V, \mathcal{E})$ (where $|V| = n$ and $|\mathcal{E}| = E$), nonnegative edge weights, and k-pairs of vertices, find the shortest path between each of these pairs.

The best-known classical implementations for *exact* solutions to the shortest paths problem take $\tilde{O}(kE)$ time to compute shortest paths between k pairs of vertices, and $\tilde{O}(nE)$ time to compute shortest paths between all pairs of vertices [31]. Recent methods using matrix multiplication to solve all-pairs shortest paths take time $\tilde{O}(M(n))$ (for graphs with unweighted edges) [60] and time $O((nW)^{2.688})$ (for graphs with maximum edge weight W) [8], where $M(n)$ is the time for matrix multiplication (currently known to be $o(n^{2.376})$). We further discuss the history of this problem in Section 4.5.

We present a $O(\log n)$ times optimal approximation algorithm for the k-pairs shortest paths problem on undirected graphs with nonnegative edge weights that runs in $\tilde{O}(E + k)$ time. Thus we can find approximate shortest paths for *all* pairs in $\tilde{O}(E + n^2) = \tilde{O}(n^2)$ time.

[1] $\tilde{O}(t)$ time is $O(t \log^{O(1)} n)$ time. (We use the standard $\tilde{O}$ notation for cleaner statement of bounds in the introduction. The technical sections give all the exact bounds later.

A tradeoff is also presented between quality of the approximation and running time. We obtain a $32d$ approximation to k-pairs shortest paths in time $O((Ed + nd^2)n^{2/d} \log n + k\beta n^{1/d} \log\log n)$. This means that we can achieve a better running time than known exact algorithms even for paths that are only a constant factor times the length of the shortest path.

One can remove a factor of $\log n$ in this tradeoff in the case of *all pairs* by constructing a sparse average cover, instead of a sparse neighborhood cover (i.e. constructing only one layer of the layered sparse neighborhood cover). In the case of all pairs, E. Cohen has recently pointed out that a pairwise cover will suffice, and this further improves the constants in the tradeoff. See her paper [30].

Further applications of the data structure. This chapter shows how to use sparse neighborhood covers for approximating k-pairs shortest paths. Rao [59] also uses the near-linear construction of sparse neighborhood covers presented here and in [13] to find small edge cuts in planar graphs. Sparse neighborhood covers seem to provide a rich representation of the local neighborhoods in a graph: for further distributed and sequential applications see [13].

4.1.1 Structure of this chapter

In Section 4.2, an overview of the new fast algorithm to construct the sparse neighborhood covers data structure is presented. A formal presentation of the algorithm that constructs the data structure is given in Section 4.3, and the analysis of the algorithm is in Section 4.4. Finally Section 4.5 shows how to approximate k-pairs shortest paths using sparse neighborhood covers.

4.2 Overview of the construction

This section presents an overview of the algorithm to which constructs a sparse neighborhood cover. Code, details, and analysis follow in Sections 4.3 and 4.4. *For clarity of exposition, we will assume throughout this overview section that we have set* $d = \log n$. Section 4.3 generalizes this to general d.

The algorithm operates in phases: each phase produces a subcollection $\mathcal{Y}$ of sets that will be placed in the final cover $\mathcal{X}$. Let R be the set of nodes in the current phase whose λ-neighborhood is not entirely contained in some set in the partially constructed cover $\mathcal{X}$. Then the sets in $\mathcal{Y}$ will have the following special properties:

1. (Low diameter) For all $Y \in \mathcal{Y}$, Diam$(Y) = O(\lambda \log n)$

2. (Overlap) Each node is in at most one set Y in $\mathcal{Y}$

3. (Neighborhoods Covered) A constant fraction of the nodes $u \in R$ will have their λ-neighborhood contained in some set $Y \in \mathcal{Y}$.

Clearly, if we can construct such a subcollection $\mathcal{Y}$, in $O(\log n)$ phases all nodes will have their λ-neighborhoods covered in some set $Y \in \mathcal{X}$, and we will have constructed a sparse neighborhood cover. In order to keep in mind "the big picture" for the construction, it helps to recall the simple greedy algorithm for constructing a weak network decomposition that appeared in Section 1.2.2. This new algorithm will also iteratively grow an individual set to be placed in the cover in layers, until a set of stopping conditions is reached. The following table shows the difference between the two procedures in the context of a unified structural framework. We follow this framework in the exposition that follows. Section 4.2.1 describes how the algorithm will construct a single Y in the cover, Section 4.2.2 discusses stopping conditions, and Section 4.2.3 shows how new non-overlapping sets Y grow around existing Y in the collection $\mathcal{Y}$. Again the code and formal analysis appear in Sections 4.3 and 4.4.

	simple greedy	**cover greedy**
how to grow	interior expands so as to fill border; new border explored	$\psi(Y)$ expands to $\psi(Z)$, Y expands to Z; new Z and $\psi(Z)$ explored.
when to stop	more in interior than border	more in $\psi(Y)$ than $\psi(Z)$ *and* in Y than Z *and* 3rd stopping condition.
how to grow around existing sets	don't enter a previous cluster's border	don't enter a previous cluster's set Y and $\psi(Z)$ won't enter a previous cluster's $\psi(Z)$.
when color (or subcover) is done	all nodes in some interior or border	all nodes in some $\psi(Z)$

4.2.1 Growth of a single set

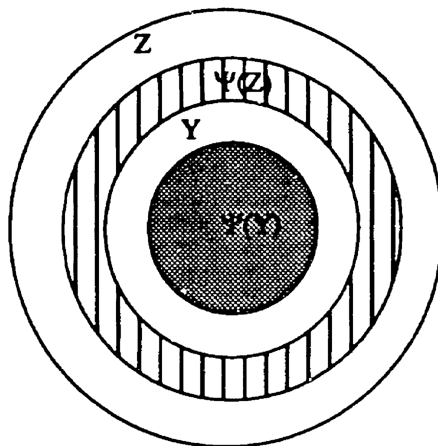

Figure 4-1: Cluster 1 is grown iteratively; Y, $\Psi(Z)$, and Z all grow in successive bands of width λ around $\Psi(Y)$. (Note that Y, $\Psi(Z)$ and Z in this picture are balls, not empty rings, and contain by definition all nodes within their borders.) If stopping conditions are not met, at the next level Y becomes the old Z, $\Psi(Y)$ grows to $\Psi(Z)$ and the new Z contains the (2λ) neighborhood of the new Y. $\Psi(Z)$ consists of those nodes whose λ-neighborhood lies in Z. The stopping conditions are guaranteed to be met in $O(\log n)$ levels.

We show how to grow a single set Y for the cover. Y is grown iteratively, in layers, and outer layers form something of a shield around Y. In particular, Y is built in a BFS fashion around a center vertex v. Y is initially defined to be the λ-neighborhood of v. We keep track of four layers around the "internal kernel" called $\Psi(Y)$ or the *interior* of Y, which is initially just the center vertex v. $\Psi(Y)$ consists of those nodes whose λ-neighborhood is fully subsumed in the set Y; Z is the 2λ neighborhood of Y; and $\Psi(Z)$ consists of those nodes whose λ-neighborhood is subsumed by Z (see Figure 4.2.1). Note that $\Psi(Y) \subset Y \subset Z$ and $\Psi(Y) \subset \Psi(Z) \subset Z$. We will sometimes refer to the set Z as the set Y's associated *cluster*.

As stated above, Y starts growing from a single node: $\Psi(Y)$ is initially just the single node v, and Y is its λ-neighborhood. If a set of stopping conditions are not met, at the next iteration, $\Psi(Y)$ becomes the old $\Psi(Z)$ and Y becomes the old Z, and the algorithm does breadth first search to construct a new Z which contains the (2λ)-neighborhood of the new Y.

4.2.2 When to stop

Recall R is the set of nodes whose λ-neighborhood is not yet entirely contained in any of the sets in the subcover constructed so far. We now describe the three stopping conditions, that govern when the growth procedure described above will terminate. One of the stopping conditions says

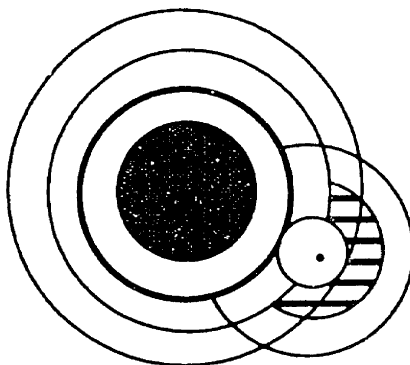

Figure 4-2: Cluster 2 begins growing with $\Psi(Y)$ any single node outside previous $\Psi(Z)$. Notice this ensures that Y, the r-neighborhood of $\Psi(Y)$, will not overlap with previous Y's. Previous Y's in fact form an impenetrable barrier (bold line) that nothing else can enter. In addition, Cluster 2's $\Psi(Z)$ (striped region) does not enter previous $\Psi(Z)$.

that the number of nodes (in R) in $\Psi(Y)$, the interior of the set, must be a constant fraction of the nodes (in R) in $\Psi(Z)$. Another says that the number of nodes (in R) in Y must be a constant fraction of the nodes (in R) in Z. These two conditions will help guarantee that the cover is sparse, and this will be discussed further in Section 4.2.3. Here, we only remark that these stopping conditions are guaranteed to be met in $O(\log n)$ levels, because each time one is not satisfied, the diameter of the interior set grows by that constant fraction. Since this can only happen $O(\log n)$ times for each before we exceed n, the diameter of any Y will be at most $O(\lambda \log n)$.

Another stopping condition is concerned with making sure that the BFS exploration of the outer Z layer does not involve too much computation. Notice that computation in $Z \setminus Y$ is in some sense "wasted" BFS computation, since these nodes are not then placed in the cover. It is therefore crucial that we can prevent the Z layer from performing huge BFS forays over the same edges for many different clusters. Thus, a cluster will always expand a layer (make Z into the new Y) if it has done too much exploration in the outer Z layer. (See Section 4.4.2 for details). We will show that this stopping condition is also guaranteed to be met in $O(\log n)$ iterations.

4.2.3 How to grow around existing sets

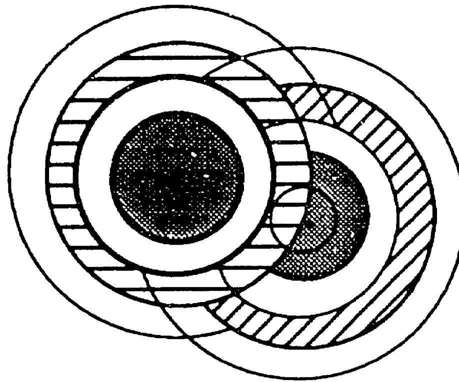

Figure 4-3: The second layer of Cluster 2 (i.e. the stopping conditions were not met in the first layer.) Notice that the new $\Psi(Y)$ does not contain all of Y from the previous layer, since, $\Psi(Y)$ is the old $\Psi(Z)$, which does not extend through a previous cluster's $\Psi(Z)$. Even though Y and Z do not extend into the territory of a previous cluster's Y layer, Y contains the λ-neighborhood of $\Psi(Y)$ and Z contains the λ-neighborhood of $\Psi(Z)$.

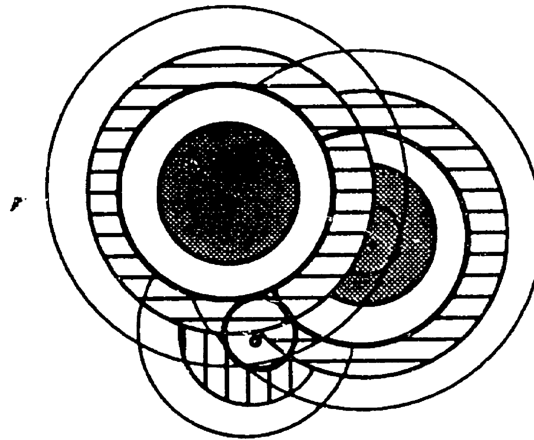

Figure 4-4: The third cluster begins to grow. As before, the kernels $\Psi(Y)$ are shaded, the sets Y are marked with a bold line, $\Psi(Z)$ is striped, and Z is the outside ring. Ignore for a moment when a node participates in the outer Z ring; then every node participated in the BFS's at most twice: a node can lie in at most one set Y, and prior to being placed in Y, it could lie in at most one $\Psi(Z)$. The BFS done in the Z layer, we end up charging to the Y cluster for which it was grown. When subcollection $\mathcal{Y}$ is complete, every node appears in a $\Psi(Z)$, for some cluster Z, and the stopping conditions ensure that a constant fraction of the nodes appear in $\Psi(Y)$ for some Y. Thus every node appears in at most one Y, and a constant fraction have their λ-neighborhoods contained in Y as well.

In this section, we show how a set Y and its cluster grow around previous non-overlapping Y from the same subcollection $\mathcal{Y}$ in the cover $\mathcal{X}$. A new cluster is always grown starting from a node v which lies outside $\Psi(Z)$ of any previous Y. Notice that this insures that the r-neighborhood of v lies outside any previous Y.

When a new cluster then grows into territory already occupied by a previous cluster, it's BFS growth is curtailed by the previous cluster's layers. Each of the successive shields around an existing cluster permit a different degree of penetration: namely no cluster is allowed to grow into a previous cluster's Y (this will ensure that the sets put in $\mathcal{Y}$ are disjoint). (See Figure 4.2.3.) A cluster's $\Psi(Z)$ does not grow into a previous $\Psi(Z)$, and the Z level is entirely permeable. (We control the cost of repeated BFS forays into previous Z instead by a stopping condition on the growth of a single cluster; see Section 4.4.2.) We point out, to aid comprehension, that the relation $Y \subseteq \Psi(Z)$ may not hold for clusters grown after the first cluster (see Figure 4.2.3).

Figures 4.2.3 - 4.2.3 give snapshots of how new clusters grow. We stop growing clusters in the subcollection Y when every node is in $\Psi(Z)$ for some cluster Z.

This ends the overview of the construction. We present the algorithm more formally (and for general d) in the next section, and then the proofs of correctness and the complexity analysis follow in Section 4.4.

4.3 Formal description of the algorithm

In this section, the algorithm `All_Cover`, is introduced. Given d and λ, `All_Cover` constructs a sparse (χ, d, λ)-neighborhood cover (see Section 1.2.4 for the definition.) Recall that in the reduction in Section 4.5, d is fixed to control the quality of the approximation (and we typically set $d = \log n$), and then `All_Cover` is called with $O(\log n)$ different λ's to produce $O(\log n)$ different sparse neighborhood covers.

We remark that `All_Cover` is actually building a *sparse tree cover* (see Section 4.5 for the definition) because it builds a BFS tree from a node in each set in the cover.

`All_Cover` is built from two intermediate sub-procedures: Procedure `Cluster` that grows a single set to be placed in the cover, and Procedure `Cover` that calls `Cluster` to produce a sub-collection $\mathcal{Y}$ of the sets in the Cover $\mathcal{X}$.

```
Ψ(Z) ← {v}
Z ← {N_λ(v)}
repeat
   Ψ(Y) ← Ψ(Z)
   Y ← Z
   Perform a multi-origin BFS w.r.t. Y to depth 2λ in G(V \ ⋃𝒴)
   Add all nodes encountered to Z
   Ψ(Z) ← {v | v ∈ U ∩ Z, dist(v,Y) ≤ λ}
until |Z| ≤ n^{1/d} · |Y|
   and |Ψ(Z)| ≤ |R|^{1/d} · |Ψ(Y)|
   and Deg(Z) ≤ n^{1/d} · Deg(Y)
return (Ψ(Y), Y, Ψ(Z), Z)
```

Figure 4-5: Procedure $\texttt{Cluster}(R, U, v, \mathcal{Y})$.

```
U ← R
Ψ(R) ← ∅.
𝒴, 𝒵 ← ∅.
while U ≠ ∅ do:
   Select an arbitrary node v ∈ U.
   (Ψ(Y), Y, Ψ(Z), Z) ← Cluster(R, U, v, 𝒴)
   Ψ(R) ← Ψ(R) ∪ Ψ(Y).
   U ← U \ Ψ(Z).
   𝒴 ← 𝒴 ∪ {Y}.
   𝒵 ← 𝒵 ∪ {Z}.
end-while
return (Ψ(R), 𝒴, 𝒵)
```

Figure 4-6: Procedure $\texttt{Cover}(R)$

4.3.1 Procedure Cluster

Procedure **Cluster** constructs a single set Y which will be placed in the cover. Y is built around a center vertex v. At all times, **Cluster** keeps track of four layers around an "internal kernel" called $\Psi(Y)$. $\Psi(Y)$ consists of those nodes whose d-neighborhood is fully subsumed in the set Y; Z is the $2d$th neighborhood of Y; and $\Psi(Z)$ consists of those nodes whose d-neighborhood is subsumed by Z (see Figure 4.2.1).

4.3.2 Procedure Cover

The role of this procedure is to construct a partial cover for the neighborhoods of the nodes

```
R ← V                          /* R is the collection of remaining (unsubsumed) balls */
X ← ∅                                               /* X is the output cover */
repeat
   (Ψ(R), Y, Z) ← Cover(R)                          /* invoke procedure Cover */
   X ← X ∪ Y
   R ← R \ Ψ(R)
until R = ∅
return X
```

Figure 4-7: Algorithm All_Cover.

in the set R. Throughout, the procedure maintains the set of "remaining" nodes U. This set is a subset of R, containing those nodes from R whose neighborhoods have not yet been subsumed by the constructed cover. Initially $U = R$, and the procedure returns once $U = \emptyset$.

4.3.3 Covering all neighborhoods

This section describes the algorithm **All_Cover**, whose task is to construct a sparse neighborhood cover, i.e., a cover with low maximum degree. The input to the algorithm is a graph $G = (V, \mathcal{E})$ (where $|V| = n$ and $|\mathcal{E}| = E$) and integers $\lambda, d \geq 1$. The output collection of cover clusters, $\mathcal{X}$, is initially empty. The algorithm maintains the set of "remaining" nodes R. These are the nodes whose neighborhoods are not yet subsumed by the constructed cover. Initially $R = V$, and the algorithm terminates once $R = \emptyset$. The code for algorithm **All_Cover** appears in Figure 4-7.

The algorithm operates in at most $d \cdot n^{1/d}$ phases. Each phase consists of the activation of the procedure Cover(R), which adds a subcollection of output clusters $\mathcal{Y}$ to $\mathcal{X}$ and removes from R the set of nodes $\Psi(R)$ whose r-neighborhood appears in some $Y \in \mathcal{Y}$. Here is where we make use of the property of procedure **Cover** that the partial covers $\mathcal{Y}$ constructed by it consist of *disjoint* clusters. This guarantees that each phase of the algorithm contributes at most one to the degree of each vertex in the output cover $\mathcal{X}$.

The next section proves the correctness of the above procedures and gives the complexity analysis.

4.4 Analysis of the algorithm

In this section, we prove the following theorem:

Theorem 4.4.1 *Given a graph $G = (V, \mathcal{E})$ (where $|V| = n$ and $|\mathcal{E}| = E$), a cover $\mathcal{N}_r(V)$, and integers $d, \lambda \geq 1$, algorithm* `All_Cover` *described in Figure 4-7 constructs a (d, λ)-neighborhood cover $\mathcal{X}$ which satisfies the following:*

1. *$\forall v \exists X \in \mathcal{X}$ such that $N_\lambda(v) \subset \mathcal{X}$.*
2. *$Diam(\mathcal{X}) = O(d\lambda)$,*
3. *Procedure* `Cover` *is invoked $O(dn^{1/d})$ times, and*
4. *$\chi = O(dn^{1/d})$.*

Furthermore, the running time of algorithm `All_Cover` *is $O((Ed + nd^2)n^{2/d})$.*

4.4.1 Correctness

To verify that we have produced a sparse (χ, d, λ)-neighborhood cover, we must check three things: that we have covered all λ-neighborhoods, that the sets have low diameter, and that each node is not in too many sets.

Low diameter Recall the three stopping conditions on the growth of a cluster. For the first or second stopping conditions to fail to be met, this means that $|Z| \geq n^{1/d} \cdot |Y|$ in the case of the first stopping condition or $|\Psi(Z)| \geq |R|^{1/d} \cdot |\Psi(Y)|$ in the case of the second. Since each time the stopping condition fails, we increase what's inside by a factor of $n^{1/d}$, each of these stopping conditions can fail at at most d incremental stages. The third stopping condition says that $Deg(Z \setminus Y)$ is less than or equal to $n^{1/d}$ times $Deg(Y)$, and since the sum of the degrees of the entire graph is at most $E \leq n^2$, the third stopping condition can fail at at most $2d$ incremental stages. Therefore the number of iterations of the growth process for which *any* of the stopping conditions fail is at worst $4d$, and so for one of the first $4d + 1$ steps at which we might grow a cluster *all* three stopping conditions will hold, and the cluster will stop growing. Since each time we grow Y, we add an addition distance λ to the radius, the diameter of the sets is bounded by $O(\lambda d)$.

Covers and is sparse The fact that the algorithm constructs a cover and that it is sparse follow from properties of subcollection $\mathcal{Y}$: We first show that the sets Y placed in each $\mathcal{Y}$ are non-overlapping. Because a new center is always picked outside of any previous cluster's $\psi(Z)$, we know its λ-neighborhood will be distance at least 1 away from any previous cluster's Y (since $\psi(Z)$ provides a buffer zone of size λ.) Then $\psi(Z)$ will not enter into a previous cluster's $\psi(Z)$, so $\psi(Z)$ is at least λ away from any previous cluster's Y. So if the new set Y grows so that $\psi(Z)$ becomes $\psi(Y)$, $N_\lambda(\psi(Y))$ will not intersect any previous Y. So the sets in the layer $\mathcal{Y}$ are non-overlapping.

Therefore, the number of sets each node is contained in, is at most the number of subcollections $\mathcal{Y}$ we need to construct in order to cover the λ-neighborhoods of every node. When a cluster Y stops growing, the stopping conditions guaranteed that a $n^{1/d}$ fraction of the remaining nodes U whose λ-neighborhood nodes in $\Psi(Z)$ were in $\Psi(Y)$ for each set Y. Since every node lies in $\Psi(Z)$ for some $Y \in \mathcal{Y}$, and the sets $\Psi(Y)$ are disjoint, this means that in each $\mathcal{Y}$ at least a $n^{1/d}$ fraction of the remaining uncovered nodes have their λ-neighborhoods placed in some set in the cover. Thus in d phases $\mathcal{Y}$, *every* node has its λ-neighborhood covered, and so each node appears in at most $dn^{1/d}$ sets and the cover is sparse.

4.4.2 Complexity analysis

W now argue that the cover is constructed in near-linear time. We first present the argument for $d = \log n$, for clarity.

To construct $\mathcal{Y}$, we bound the cost of checking the stopping conditions, and then performing the appropriate BFS explorations. The stopping conditions need to be checked at most $O(\log n)$ times for each cluster, because as we argued above, the conditions will be met in $O(\log n)$ levels. To check the stopping conditions, we count nodes in $Z, \forall Z \in \mathcal{Z}$. The second stopping conditions insures that for all Z the number of nodes in Z is less than a constant fraction times the number of nodes in Y, so $O(\sum_{Z \in \mathcal{Z}} |Z|) = O(\sum_{Y \in \mathcal{Y}} |Y|) = O(n)$, since the Y are disjoint. Thus the cost of checking the stopping conditions to construct $\mathcal{Y}$ is $O(n \log n)$.

Now consider the cost of the BFS. Ignore for a moment the cost of the BFS exploration of the final layer $Z \setminus \Psi(Z)$, for all clusters in $\mathcal{Y}$. Then each node participates at most twice in BFS's during the construction of $\mathcal{Y}$: each edge is placed in at most one set Y, and prior to being placed in Y, it could lie in at most one $\Psi(Z)$. Thus without the Z layer, each edge is

explored twice, for a total cost of $O(E)$.

Now we bound the complexity of examining edges in Z. Notice that when $\lambda = 1$ this is trivial, since every edge in $Z \setminus \Psi(Z)$ has an endpoint in $\Psi(Z)$, and every node is in a unique $\Psi(Z)$. For $\lambda > 1$, bounding the amount of work done in Z is controlled by means of another stopping condition on the growth of individual clusters. Namely, if we have done too much work on exploring the layer $Z \setminus Y$, we are then obligated to grow Y to contain Z.

More specifically, define $deg(v)$ to be the degree of node z in the graph G. Let $Deg(V) = \sum_{v \in V} deg(v)$. Now, we use the final stopping condition on clusters to ensure $Deg(Z \setminus Y)$ is less than or equal to a constant fraction times $Deg(G) = 2E$. Together with the stopping condition on the size of Z, this gives $\sum_{Y \in \mathcal{Y}} |Z \setminus Y| \leq 2cE$, $c > 0$. Therefore, the cost of the BFS exploration of $Z \setminus \Psi(Z)$ is also $O(E)$.

Thus the total time to construct subcollection $\mathcal{Y}$ is $O(n \log n + E)$. Constructing $O(\log n)$ sub-collections $\mathcal{Y}$ to complete the cover $\mathcal{X}$, the following corollary is obtained.

Corollary 4.4.2 *The algorithm described constructs a* $(\log n, \log n, \lambda)$*-sparse neighborhood cover in time* $O(E \log^2 n + n \log^3 n)$

To extend to general d, the cost of checking all the stopping conditions will be $E + dn$ as before, they could fail $O(d)$ times, for a total cost of $(E+nd)d$. Then, as before, we must bound the complexity of examining edges in Z. By the third stopping condition, $\sum_{Y \in \mathcal{Y}} |Z \setminus Y| \leq O(n^{1/d}E)$, and we construct $n^{1/d}$ sub-collections $\mathcal{Y}$. Thus the complexity bound in Theorem 4.4.1 follows. □

4.5 Approximating k-pairs shortest paths

In this section we introduce an algorithm for approximating the k-pairs shortest paths problem on undirected graphs with nonnegative edge weights in time $O(E \log^2 n + n \log^3 n + k(\log n \log \log n))$. We achieve the substantial improvement in running time over known algorithms for this problem by reducing the approximation of k-pairs shortest paths to sparse neighborhood covers. The bottlenecks in the running time of the exact algorithms come either from repeated application of Dijkstra's single-source shortest paths algorithm, or from matrix multiplication. We are able to bypass the bottlenecks of both these approaches at the cost of

producing an approximate solution. We are thus able to substantially improve upon the time to compute k-pairs shortest paths to $\tilde{O}(E + k)$ (approximate).

We have already noted the best-known bounds for exact k-pairs shortest paths above. More precisely, the algorithms that achieve these bounds run in time $O(kn \log n + kE)$ with a Fibonacci heap implementation [36] and $O(kE \log n)$ time with a binary heap implementation [31] of the priority queue used by Dijkstra's algorithm. For the all-pairs shortest paths problem, the time bounds are $O(n^2 \log n + nE)$ and $O(nE \log n)$, respectively [31, 36].

There have been other improvements on the running time for the all-pairs shortest paths problem for some special case graphs [63, 42, 26, 37, 6, 38, 43, 55, 41], but they all have worst-case time $O(nE)$. In fact, Karger et al. [43] show that $\Omega(nE)$ is a lower bound on all these existing algorithms, which are "path-comparison based".

As for the approaches to (exact) all-pairs shortest paths that rely on matrix multiplication, Alon et al. [8] describe an $O((nW)^{2.688})$ time algorithm for the case of integer edge weights whose absolute value is less than W. Recently, Seidel [60] has improved this result to give an algorithm for the unweighted case which runs in time $O(M(n) \log n)$, where $M(n)$ is the time for matrix multiplication (currently known to be $o(n^{2.376})$). Seidel also mentions in [60] that Alon, Galil, and Margalit have communicated to him that they have in turn improved his result, solving exact all-pairs shortest paths in the case of integer edge weights between 0 and W in $O(W^2 M(n) \log n)$ time. Notice that the algorithm presented in this paper runs faster than these recent algorithms in both the unweighted and weighted cases [8, 60]: we bypass the bottleneck of the new matrix multiplication methods at the cost of producing an approximate solution.

There has also been recent work on approximation algorithms for shortest paths in the domain of parallel computation. Klein [45] extended the work of [65] to the weighted case, obtaining a randomized PRAM algorithm that can $1 + \epsilon$ approximate shortest paths between k pairs of vertices in $O(\sqrt{n}\epsilon^{-2} \log n \cdot \log^* n)$ time using $(kE \log n)/\epsilon^{-2}$ processors (where ϵ is constant). When this algorithm is run sequentially, the time is not as good as known sequential algorithms for the exact problem. Hence, Klein's approximation algorithm is strictly of interest for parallel computation.

Recently, E. Cohen has also considered approximating shortest paths in parallel. See [30].

4.5.1 Tree covers

The explanation of the algorithm is made simpler through the notion of *sparse tree covers*.

Definition 4.5.1 *For an undirected graph $G(V, \mathcal{E})$, a $(\chi, d, \lambda,)$*-tree cover *is a collection $\mathcal{F}_{d,\lambda}$ of trees in G, that satisfies the following properties:*

1. *Every vertex is in at most χ sets.*

2. *Every tree $F \in \mathcal{F}_{d,\lambda}$ has depth $\leq O(d\lambda)$.*

3. *For every two vertices $u, v \in V$ whose distance in G is λ or less, there exists a common tree $F \in \mathcal{F}_{d,\lambda}$, containing both.*

A (χ, d, λ)-tree cover is said to be sparse, *if χ is at most $dn^{1/d}$.*

Note that the algorithms in Section 4.3, described exactly how to construct sparse tree covers, since we build a BFS tree that spans each set in the cover. Recall, the construction takes $O((Ed + nd^2)n^{2/d})$ time, which is $O(E\log^2 n + n\log^3 n)$ when we set $d = \log n$.

4.5.2 The algorithm

We first give our algorithm for approximating k-pairs shortest paths with unit edge weights, and later explain how to extend this to the case when the edge weights are non-negative.

Let $\delta = \log Diam(G)$. For every level $1 \leq i \leq \delta$, construct a sparse tree cover $\mathcal{F}_{d,2^i}$ for G, and number the trees in the cover. For every vertex v, and for every level i, store (in order of increasing tree number ID) a list of all trees $F \in \mathcal{F}_{d,2^i}$ that contain v. Recall that by the sparsity property, for a given level i, every vertex belongs to at most $dn^{1/d}$ different trees in $\mathcal{F}_{d,2^i}$.

Now for any given pair of vertices u, v, we want to use this as a data structure to quickly find a shortest path between these vertices. Perform a binary search over the levels $1 \leq i \leq \delta$: for a given level i, if there exists a tree $F \in \mathcal{F}_{d,2^i}$ that contains both u and v, restrict the search to lower values of i; otherwise, restrict the search to higher values of i.

The binary search will produce the minimum level i' such that there exists a tree $F \in \mathcal{F}_{d,2^{i'}}$ that contains both u and v. The algorithm returns the value $16d2^{i'}$ as the approximate shortest path between u and v.

Iterate the binary search for additional pairs of vertices.

The weighted case. The algorithms for sparse neighborhood covers presented in this paper can easily be modified to produce a *weighted* tree cover by forming shortest path trees rather than BFS trees. The algorithm presented in this section can in turn be trivially modified to handle weights by instead of having path lengths of at most $Diam(G)$, we now have a maximum path weight of $W \cdot Diam(G)$, where W is the maximum edge weight in G. The number of levels would therefore be $\delta = \log(W \cdot Diam(G))$. As we will see below, the running time of our algorithms remains asymptotically unchanged when the edge weights are polynomial.

Producing the path. The algorithm can easily be modified to return the unique path connecting a pair u, v in the common tree, in time order of the path's length.

4.5.3 The analysis

First we wish to prove that our algorithm is an $O(d)$ approximation algorithm.

Lemma 4.5.2 *For each of the k pairs of vertices given, our algorithm returns a path* length *between them within $32d$ times the length of the shortest path.*

Proof We will argue this for a given pair u, v. Note that by Definition 4.5.1, if $dist(u,v) \leq 2^i$, then the tree cover of level i has a tree containing both u and v. Since i' is the minimum level i for which this is so, it must be that $2^{i'-1} < dist(u,v) \leq 2^{i'}$. Also by Definition 4.5.1, the common tree has depth at most $8d \cdot 2^{i'} < 16d \cdot dist(u,v)$. That is, our algorithm returns the maximum length of the unique path connecting u and v in the common tree: $16d \cdot 2^{i'} < 32d dist(u,v)$. □

Now we address the running time of our algorithm.

Lemma 4.5.3 *Our algorithm for approximating k-pairs shortest paths takes $O((Ed + nd^2)n^{2/d} \log n + k(dn^{1/d} \cdot \log\log n))$ time; that is, $O((Ed + nd^2)n^{2/d} \log n)$ time to set up the tree cover data structures, and $O(dn^{1/d} \cdot \log\log(Diam(G)))$ time to answer any query.*

Proof We have already noted that the algorithm for sparse neighborhood covers in this paper can produce the tree cover as it goes along at no additional payment in time; thus the time bound for setting up a tree cover data structure follows from Theorem 4.4.1. Observe that we actually set up $\log n$ different tree covers, one for each level, which adds a factor of $\log n$ to the running time.

As for the bound on a single query, the binary search takes time $O(\log\log(Diam(G)))$. And for a given level i, a common tree can be found in $dn^{1/d}$ time by searching the ordered lists of tree ID's to which the two vertices belong. □

Lemma 4.5.4 *The nonnegative weighted version of our algorithm for approximating k-pairs shortest paths takes* $O((Ed + nd^2)n^{2/d}\log(Wn) + k(dn^{1/d} \cdot \log\log(Wn)))$ *time.*

Proof sketch: Similar to the proof of Lemma 4.5.3 with $dist(u, v)$ modified to be $w(u, v)$. □

Lemmas 4.5.2, 4.5.3, and 4.5.4 yield the following theorem for polynomial edge weights.

Theorem 4.5.5 *The approximation algorithm for k-pairs shortest paths takes time* $O((Ed + nd^2)n^{2/d}\log n + k(dn^{1/d} \cdot \log\log n))$, *and returns a solution that is within* $O(d)$ *of the exact one. Setting* $d = \log n$, *our algorithm gives an* $O(\log n)$ *approximation to k-pairs shortest paths in* $O(E\log^2 n + n\log^3 n + k(\log n \cdot \log\log n))$ *time.*

Corollary 4.5.6 *Our algorithm for approximating* all-*pairs shortest paths takes time* $O((Ed + nd^2)n^{2/d}\log n + n^2(dn^{1/d} \cdot \log\log n))$, *and returns a solution that is within* $O(d)$ *of the exact one. Setting* $d = \log n$, *our algorithm gives an* $O(\log n)$ *approximation to all-pairs shortest paths in* $O(E\log^2 n + n\log^3 n + n^2(\log n \cdot \log\log n))$ *time.*

Chapter 5

A Wait-Free Symmetry Breaker, and a Solution to the Generalized Dining Philosophers Problem

The previous chapters have all been concerned with *network decomposition*, which was primarily employed to represent local neighborhoods of a graph up to a logarithmic distance. When Awerbuch et. al. introduced network decomposition in 1989 (see [15]), the goal of that paper was actually a fast deterministic solution to the maximal independent set problem in a distributed network.

In this chapter, we come full circle, and look again at the problem of constructing a maximal independent set in a distributed network, only we will allow randomization. Furthermore, our algorithm will be proved correct in an entirely asynchronous environment, as well as robust against several different types of network faults. For example, in the course of this chapter, we introduce a measure of local dependency that captures how far a neighborhood from a node, might affect its behavior. An algorithm is called *k-wait free* if the behavior of a process in the network is only dependent on its neighbors *a constant k* distance away. The intuition is that slow or faulty nodes far away in the network do not interfere with local performance.

The principal result of this chapter is a randomized 2-wait-free algorithm that computes a maximal independent set in an entirely asynchronous distributed network. This has immediate

---

This chapter describes joint work with B. Awerbuch, and M. Smith [14].

applications to *symmetry breaking* and *dining philosophers.*

5.1 Introduction

The maximal independent set problem Consider the underlying graph of the network to be the input graph, where the nodes are network processors, and two processors are connected by an edge if there is a direct communication link between them. An MIS in this graph is an independent set of processors such that any processor is in the subset, or has a neighbor in this subset. (A *maximal* independent set, not to be confused with a *maximum* independent set, which is an independent set of maximum cardinality. Finding a *maximum* independent set is NP-Complete [40]). This paper gives a 2-wait-free algorithm for MIS which runs $O(\log n)$ expected time (see Figure 5-1).

Symmetry breaking Many natural network control operations can be reduced to the problem of finding a maximal independent set. A maximal independent set breaks symmetry, and when one can break symmetry in a network, one is able to resolve deadlocks [18], elect a leader [1], achieve mutual exclusion [34], and allocate resources [22].

Thus our efficient MIS construction is an efficient symmetry-breaker for an asynchronous distributed network. It is the first symmetry-breaker to run in poly-log time in a network without a global clock. In addition, the symmetry-breaker we present has several nice "robust against faults" properties that have been recently introduced in the distributed systems community as modern protocol design goals.

Our symmetry-breaker could be practical in a realistic distributed system, because it can tolerate faults, as captured by the following property:

Wait-freedom: The protocol tolerates ongoing faults in the sense that if some processors stop, the protocol continues to run at the same speed, i.e. the speed does not depend on slow processes (links).

We remark that the protocol presented can be coupled with a manager protocol that will periodically check and correct each link subsystem [28], so that the protocol becomes *self-stabilizing,* [32, 5, 20, 23, 54, 46] meaning it can return to normal operation in constant time, even if the network crashes, and is brought back up in an inconsistent initial state. Since the

inter-processor dependency exhibited by wait-free algorithm is local, it is easy to detect and correct initialization faults. In other words, wait-freedom is an important property for designing fast self-stabilizing algorithms. This result is due to [14] and is beyond the scope of this thesis.

Wait-freedom. Wait-freedom is an important property of distributed algorithms that has been traditionally defined in the shared memory model of distributed computing. In that model, an algorithm is said to be *wait-free* if processes will continue their operations regardless of the failure of other processes. That is, the delay between steps in an individual process is only dependent on the delay of its links to the shared memory.

We will define the same term for the message passing model of distributed computing in a way that we think captures an important metric for performance and fault tolerance in this model. Intuitively, we say an algorithm is *k-wait-free* in the message passing model, if progress of an individual process is only dependent on information from other processes distance k away. Our goal is to make k as small as possible. We think this is an important performance metric because it means a process only has to wait at most k times the maximum link delay before a step in its execution. It is also a fault tolerance metric because it means that a process can only be affected by faults that are at most distance k away. Another way to look at k-wait-freedom is that it captures the notion that the operation of a process should not be slowed or stopped by a very slow or failed process or link that is greater than distance k away.

5.1.1 Previous results

Randomized solutions to the distributed MIS problem which run in logarithmic expected time have been found by Karp-Widgerson [44] and Luby [51], *in a properly-initialized synchronous model*. The protocols of Karp-Widgerson and Luby run in set rounds, which assumes the existence of a global clock, and a consistent initial state. In an *asynchronous* model, the best known algorithms for constructing an MIS use the "synchronizers" introduced by Awerbuch [10] and the algorithms of [44] and [51].

In the *asynchronous* model, if the network graph is static and all the processes "wake up" at the same time then the best known algorithms for constructing an MIS use the α "synchronizer" introduced by Awerbuch [10] and the algorithm of [44] or [51]. The α synchronizer requires time linear in the diameter D of the network to set up. In the case where the network is

changing dynamically, the α synchronizer does not just add to setup, but may add a delay of $O(D)$ per step of the protocol where D is diameter of the network. Therefore, a protocol that uses the α synchronizer plus the protocol of Luby or Karp-Widgerson, would run in $O(D \log n)$ expected time. This added delay per step occurs because the α synchronizer may cause a wait-dependency of $O(D)$ in the protocol. The wait-dependency would also mean a processor or link failure anywhere in the network might prevent any other process in the system from making progress. Synchronizers in [10] are not self-stabilizing either; methods of [16] can achieve self-stablization at the price of increasing the time to $O(D)$.

5.1.2 Our results

We present a new MIS algorithm for symmetry-breaking, and a new dining philosophers algorithm. Our MIS algorithm runs in expected $O(\log n)$ time on a purely asynchronous network, and the dining philosophers algorithm is the first to meet the lower bound of $O(\delta)$ expected time. Our algorithms are robust in the sense that they are *2-wait-free* (i.e., processes are only dependent on their neighbors and the processes adjacent to their neighbors during the protocol).

We remark that the difficulty in the design and analysis of an asynchronous MIS protocol as opposed to the properly-initialized synchronous methods of Karp-Widgerson [44] and Luby [51], is dealing with variable delays between the links. Without a global clock, some nodes can communicate very fast and run "way ahead" of others in the protocol. We are able to overcome this difficulty and ensure some degree of fairness among all nodes competing to enter the MIS. Thus we are able to ensure logarithmic convergence.

Dining philosophers. The *Dining Philosophers* problem and its extensions, are classical problems in the theory of distributed computing [33, 53, 58, 27, 52, 22, 29]. The essence of dining philosophers is arbitrating between conflicting demands of different processes for shared resources (See Section 5.5 for definitions).

As an application to our MIS protocol, we get a new randomized algorithm for the dining philosophers problem. The best previous known algorithms for the generalized dining philosophers problem due to Awerbuch and Saks [22] (randomized), and Choy and Singh [29] (deterministic, special hardware assumptions) have $O(\delta^2)$ expected response time, where δ is

Authors	Time	Wait-dependency
[44, 51] + [10, 20]	$O(D)$	D
Algorithm in this chapter	$O(\log n)$	2

Figure 5-1: Our MIS algorithm versus existing ones; n is the total number of nodes and D is the diameter of the network.

Authors	Response time	Wait-dependency
[53]	$O(c^{\delta})$	$O(\delta)$
[27]	$O(n)$	$O(n)$
[64]	$O(\delta^{\log \delta})$	$O(\log \delta)$
[22]	$O(\delta^2)$	$O(\delta)$
This paper	$O(\delta)$	2
Lower bound	$\Omega(\delta)$	1

Figure 5-2: Our Resource allocation algorithm versus existing ones; c is the number of colors used to color the network graph, n is the number of nodes in the network, and δ is the number of conflicting jobs.

the number of competing jobs, and is also a lower bound on response time. (See Figure 5-2.) This work achieves optimal $O(\delta)$ expected response time in the general model. In addition, our solution is 2-wait-free.

5.1.3 Structure of this chapter

We present the model in Section 5.2. We present the asynchronous protocol in Section 5.3; the proofs of correctness and convergence follow. In Section 5.4 we give a careful analysis of the algorithm, and show we can expect it to run in $O(\log n)$ time, even with a malicious adversary with full control over the timing of all link delays in the protocol. Finally, Section 5.5 presents the application to efficient randomized Dining Philosophers.

5.2 The model and the problem statement

The following is a sketch of the standard model. For a more formal treatment, where each process is modeled as a probabilistic, timed I/O automaton, the reader is referred to [61].

The asynchronous network model. We use the standard asynchronous model of a distributed system, where the link delays are finite but unpredictable, and there is no common

memory that is shared by the processes. (We do assume FIFO, in other words if on the link from i to j, process i sends message(1) and then message (2) to j, then j will receive first message(1) and then message(2)). Also each process has its own execution speed that is unknown to its neighbors. The message delay time will be the link delay plus the time it takes the process to do any local computations to prepare the message.

Processors need *not* be assumed to have unique IDs, although we will make this assumption for clarity of exposition. We remark that it is easy in a randomized protocol to give processors unique IDs with high probability, we simply have them pick IDs uniformly at random from the set $1 \dots n^3$ and then each node will have an ID unique from its neighbors with high probability. In fact, the algorithms we present below will not even need this– we will see that we can break ties arbitrarily for nodes of identical degree, and so no IDs are needed at all.

Input/Output. The network protocol is defined by the conjunction of identical *nodal programs*, executed by all processors in the network. The input to the nodal program is the collection of adjacent network edges; the output of the nodal program is a special "flag" that is marked 1 for nodes in the MIS and "0" for nodes outside the MIS.

Message delay. Message delays during the course of the protocol are assumed to be fixed in advance, but are allowed to change arbitrarily as a function of time. The protocol does not know this message delay function. (In other words, all we are guaranteed is that if a message is sent over link ij at time t, it will arrive at time $f_{ij}(t)$, regardless of previous behavior of the protocol, but f can be an arbitrarily horrible function of time). Our time bounds will hold *for any fixed function f* randomized over only the coin tosses of the algorithm.[1]

We can state our time bounds in terms of some constant fixed upper bound on the maximal message delay on a link (based on the link delay function, but not known by the protocol in advance). Alternately, when not reasoning explicitly about exact link delays, we will use the normalized time complexity, namely physical execution time divided by maximal message delay

[1]This is sometimes called the oblivious, or non-adaptive adversary model, because we can think of f as set by an adversary, who cannot, however, see the coin tosses of the algorithm and modify link delays adaptively. We comment that such a stronger adversary, or malicious adversary, is usually considered in the context of cryptography, where adaptive attacks can be launched against the security of a cryptosystem. For modeling behavior of real distributed systems, defeating an oblivious adversary that does not see internal coinflips and plot accordingly should suffice.

on a link. This is equivalent to measuring physical time assuming message delay varies between 0 and 1 time units of some global clock [10, 39, 20].

Wait-freedom. Consider a reactive asynchronous distributed algorithm, where nodes enter the protocol and exit from the protocol in an online fashion, e.g. deadlock resolution [18], end-to-end [3, 17, 4], reset [2, 20], or dining philosophers [27, 52, 22]. Such algorithm is said to be *i-wait-free*, or *i-waiting*, if time elapsed from the entrance of a particular node into the protocol till its exit is only dependent on the speed of processors and links at distance i away. In other words, the longest "waiting chain" in the algorithm is of length i. Minimizing the length of the waiting chain has been a subject of discussion in [29], where the computational model has been strengthened by certain assumptions.

If a distributed algorithm is m-wait-free, then the time complexity of that algorithm must be multiplicative factor of m times the maximum link delay, or for the the case where link delays are normalized, times m.

Asynchronous PRAM model We remark that our algorithm is also an asynchronous PRAM algorithm, on the underlying input graph, since individual processors do only constant amount of work. The one possible difficulty, is that our graph is not necessarily constant degree: we are allowing a process to accept messages from *all* its neighbors in one step. To get around this, we could instead place a binary tree of processors for each node, where the leaves represent all its neighbors. Then, at an additional $\log n$ factor to communicate between neighbors, each node can accept a constant number of messages to determine if a higher priority neighbor is attempting to enter the MIS. Since we can charge the time it takes a message to travel through the binary tree to the asynchronous link delay function, the correctness analysis is unchanged.

5.3 The Protocol

In this section we present the asynchronous MIS protocol. We first review the elegant synchronous protocol of Luby [51], and discuss the difficulties in simulating such a protocol in an asynchronous environment.

5.3.1 Review of Luby's protocol

Luby's synchronous MIS protocol, as given in [51] proceeds in rounds. In each round, process i flips a coin c_i, where

$$c_i = \begin{cases} 1 & \text{with probability } 1/(2d(i)) \\ 0 & \text{otherwise,} \end{cases}$$

where $d(i)$ is the degree of node i in the underlying graph.

Processor i then compares the value of its coin to the coins of its neighbors, and enters the MIS if its coin is 1, and for all its neighbors such that $d(j) > d(i)$ or $d(j) = d(i)$ and $\mathrm{ID}(j) > \mathrm{ID}(i)$, j's coin is 0. Luby shows that in $O(\log n)$ expected rounds, this constructs an MIS.

5.3.2 The difficulty of asynchrony

In an asynchronous environment, it is not clear how to implement a protocol like Luby's. Without a global clock, there is no way to insure that processes flip at the same rate. If we do not control the rate a process flips as compared to its neighbors, many things can go wrong. For instance, a fast-flipping process might have multiple chances to flip a 1 and kill slower-flipping neighbors.

The crux of the problem is as follows: suppose a process i flips a 0. Then its neighbors who have flipped 1's should have a chance to enter the MIS before i flips again. A neighbor which flipped a 1 would thus like its neighbors to *freeze,* while it checks to see if it will survive and enter the MIS. Except, if we allow each of i's neighbors to freeze i in turn, i can stay frozen a long time with no chance to enter the MIS.

The solution to this in our protocol is when a process i flips a 0, it allows each neighbor to freeze it exactly once before i flips again. It is important that processor i itself chooses to freeze, based on its current information, rather than each process j can choose to freeze i in turn. When a neighbor flips again, this unfreezes i, *and a neighbor who unfreezes i's new coin cannot freeze i again until i has flipped a new coin too.* The code for the protocol follows in Figure 5-3, and the variables in the code are described in Section 5.3.3.

5.3.3 The description of the code

In the distributed system, a process will represent a node in the graph, and edges will be represented by the communication channel between processes. Initially each process is given

```
Program RECEIVE(C):                                   /* program on process i */

C = MIS(Neighbors)
  Flip-Coin

C = Query(F, j, d(j))
  SEND to j, ACK(Coin, ID, d)
  if Freeze(j) = 1, Freeze(j) ← 0
  if d_j > d and F = 1 and Coin = 1
    Coin(j) ← 1
    else Coin(j) ← 0
  if Coin = 0 and if ∀k Freeze(k) = 0 Flip-Coin

C = ACK(F, j, d(j))
  if Coin = 1
      if d_j < d_ID or F = 0, Coin(j) ← 0
      else Coin(j) ← 1
    if ∀k Coin(k) = 0, Enter-MIS
    else if ∀k Coin(k) ≠ UNSET, Flip-Coin
  if Coin = 0
    Coin(j) ← F
    if Coin(j) = 1, Freeze(j) ← 1
    else Freeze(j) ← 0
    if ∀k Freeze(k) = 0, Flip-Coin

C = Remove(j)
    update-flag(j) ← 1

C = InMIS(j)
  MIS-flag ← 0
  ∀k ∈ Neighbors, SEND Remove(ID)

procedure Enter-MIS
  MIS-flag ← 1
  ∀j ∈ Neighbors SEND InMIS(ID)

Procedure Flip-Coin
  ∀j s.t. update-flag(j) = 1,
    Neighbors ← Neighbors - {j}
  Coin(j) ← UNSET
  Freeze(j) ← UNMARKED
  Coin = 1 with probability 1/8d
       = 0 otherwise
  ∀j ∈ Neighbors
       SEND Query(coin, ID, d)
```

Figure 5-3: MIS Algorithm

its set of neighbors by some external system. Let i be the process with ID $= i$,[2] and let $d(i)$ be its degree in the network. For notational convenience we will associate with d_i the ordered pair $(d(i), i)$ and say that $d_i > d_j$ if $(d(i) > d(j)$ or $d(i) = d(j)$ and $i > j)$. The code for a process i is shown in Figure 5-3. Below we describe the internal variables used by the process.

ID: represents the process's own ID.

Coin: is the current value of i's flipped coin.

Coin(j): records information about neighbor j's coin. It has three possible values, UNSET if coin was just flipped and neighbor j's coin is unknown; 0 if ($d_j < d$ and coin $= 1$) or (j's coin $= 0$); and 1 if ($d_j > d$ and coin j's coin $= 1$) or (j's coin $= 1$ and coin $= 0$).

AckCoin(j): keeps track of last value of Coin sent to j. (We need this for self-stabilization).

Freeze(j): when coin $= 0$, this variable keeps track of whether i froze its current coin for neighbor j, has already frozen and then unfrozen its current coin for neighbor j, or has not yet frozen for j. These notions correspond to the values 1, 0, and UNMARKED respectively.

Neighbors: set of adjacent vertices not know to be in the MIS nor have a neighbor in the MIS.

Receive-ACK(j): flag that indicates whether a response to a query have been received from j (Used only for self-stabilization)

MIS-flag: flag that indicates i's status in MIS.

Below is a description of messages received by the process.

MIS(Neighbors): message from external system to begin construction of MIS.

Query($F, j, d(j)$): message from neighbor j indicating j's coin $= F$, j's ID and current degree, and also that this is a query message, requesting the value of i's coin.

ACK($F, j, d(j)$): message from neighbor j indicating j's coin $= F$, j's ID and current degree, and also that this is an ack message in response to a query.

InMIS(j): message from j saying it is in the MIS.

Remove(j): message from neighbor j saying that it has a neighbor in the MIS and should be removed from the set of active nodes.

[2] In fact, we can break ties arbitrarily by flipping a coin, and do not need unique IDs– we use them here for clarity of exposition.

5.4 Analysis of the algorithm

We first prove some essential properties of the protocol:

Lemma 5.4.1 (Safety.) *If i has MIS-flag = 1, then for all neighbors j of i, j has MIS-flag = 0.*

Proof. In the protocol a process i will join the MIS only if $\forall j$ such that $d_j > d_i$, Coin(j) = 0 and it has set Coin = 1. Assume, by contradiction, that two neighbors i and j, both enter the MIS. Let $winner_i$ be the last coin that i flipped before joining the MIS, and let $winner_j$ be the last coin that j flipped before joining the MIS. Then there must be some time t_i at which i flips the coin $winner_i$ and similarly define time t_j. Notice that by definition of the coins $winner_i = winner_j = 1$. After a process flips its winning coin, that coin will stay at 1 for all time. There are several cases.

1. t_j is before t_i, and at time t_j, $d_j > d_i$. Then at time t_i, when i queries all its neighbors, the ACK from j will say Coin $j = 1$. Since j doesn't update its degree until it flips again, and j hasn't flipped since time t_j by assumption, j's ID remains what it was at time t_j, or possibly j has entered the MIS by time t_i. In either case $winner_j$ will not allow i to enter the MIS.

2. t_i is before t_j, and at time t_i, $d_i > d_j$. Same as Case 1, by symmetry.

3. t_i is before t_j, and at time t_i, $d_i < d_j$. There are several subcases.

   (a) t_j occurs while Coin (j) is still UNSET. Then this is the same as Case 1, above.

   (b) t_j occurs after i has received an ACK from neighbor and j set its flag Coin(j), but before i has heard from all neighbors. Then i has set its flag according to the previous coin of j (which didn't win). j cannot enter the MIS until j has queried all its neighbors (including i) and received ACKs. If i has not yet heard from all its neighbors when it receives j's new query, it resets its flag Coin(j) = $winner_j$, and $winner_j$ will not allow i to enter the MIS.

   (c) t_j occurs after i has already received ACK's from all its neighbors. By assumption, all of i's neighbors of higher ID reported a flip of 0. Therefore, i has already sent an InMIS message to j which will reach j before it receives an ACK from i by the FIFO property of the links, and so j will never enter the MIS.

4. t_i is before t_j, and at time t_i, $d_j > d_i$. Same as case 3, by symmetry.

□

Lemma 5.4.2 (The Flip Again Lemma). *Let ν_j be an upper bound on the maximum link delay to distance 2 from j in the graph. Then j gets to flip again (i.e. try to enter the MIS) in time at most $5\nu_j$, if neither j or any of its neighbors have yet managed to enter the MIS.*

Proof. If coin = 1, then a process only needs to receive ACK's from all its neighbors before it can either begin executing or flip again. Thus, the delay is at most $2\nu_j$.

If coin = 0, then clearly the worst case occurs when Freeze flags get marked 1 since a process will have to wait until all these flags get marked 0 before it can flip again. Freeze flags get marked 1 only after i receives ACK's in response to Queries. If Freeze(j) got marked 1 it means j must have flipped a 1. We are interested in the case where j loses and does not enter the MIS. Freeze(j) will get marked 0 when i receives a new Query message from j or a Remove message from j. Since j flipped a 1, it will either lose, and flip again in $2\nu_j$ and thus send a Query message to i in time $3\nu_j$, for a total delay of $5\nu_j$, or j itself could enter the MIS. □

Lemma 5.4.3 (The Bounded Flips Lemma). *Suppose during an execution of the protocol, j flips at time t_0. Compare the following two events:*

$E_0 = j$ *flips a 0 at time* t_0

$E_1 = j$ *flips a 1 at time* t_0.

Then the amount of time it takes j to flip again given E_0 is always greater than or equal to the time it takes j to flip again given E_1.

Proof. Follows immediately from the protocol design: If j flips a 0 *or* a 1, j will send a message to all his neighbors, and receive one back – so far, delays are identical. At this point, if j had flipped a 1, j will flip again; if j flipped a 0, j might wait an additional delay if it is "frozen" because a neighbor flipped a 0. Since all delays are positive, the lemma follows. □

In Luby's synchronous MIS algorithm [51], Luby shows that if a node flips a 1, then the probability that any one of the neighboring nodes prevents the node from entering the MIS, is bounded by a constant. In the synchronous algorithm, each neighbor has exactly one chance to "kill" such a neighbor– because everybody flips one coin in each round, and compares. The

following lemma shows that even though there no longer are synchronous rounds, each neighbor has at most two chances to prevent a node from entering the MIS.

Lemma 5.4.4 *Suppose processor i flips 1. Processor i queries all neighbors to find out the value of their current coin. For all neighbors j, let c_j be the value of j's coin as he reports it to processor i and let $next_j$ be the value of j's coin on j's next flip. Then if c_j and $next_j$ are both 0 for all neighbors j of i such that $d_j > d_i$, then i will enter the MIS in time $2\nu_j$.*

Proof. Follows immediately from the design of the protocol. When processor i flips, i's flag Coin(j) is initially unset; however, i will set Coin(j) after i has queried j and j answers. Now, the claim is for all j, such that c_j and $next_j$ are both 0, Coin(j) must be initially set to 0, and remain 0 until i flips again or enters the MIS. If c_j is 0, by definition of c_j, i sets Coin(j) initially to 0, if j flips again and $next_j = 0$, then j will freeze for i, and not flip again until i does. So, if c_j and $next_j$ are all 0 for j with $d_j > d_i$, then Coin(j) for such j will be unset or 0, until i flips again or enters the MIS. Since i sets Coin(j) for all j before flipping again, when i sets the last Coin(j) for the slowest link j, all flags Coin(j) are set, and for all j such that $d_j > d_i$, all flags Coin(j) are set to 0. So i enters the MIS when i sets the last Coin(j) for the slowest link j, which happens within time $2\nu_j$. □

We now consider the following situation. Let processor i flip 01 in two consecutive flips. Let t_0 be the time at which processor i flips 0. We look at the execution tree, where we have a 0 branch and a 1 branch corresponding to each possible coinflip of a processor during the algorithm. We introduce the following notation. For a time t, define $f_j(t)$ to be the time at which i, got a message sent at time t, to processor j. We define $L_j(t)$ to be the maximum, over all neighbors k of j, of the delay for j to send a message to a neighbor k of j, and then receive a message back from k. $f_j(t) - t$ is the "one way" link delay from i, and L_j is the maximum two-way link delay from neighbors j of i. Notice that they depend only on the link delay function, and not on the values of coins in a particular execution. We are now ready to prove the following lemma.

Lemma 5.4.5 *Consider executions beginning at t_0 where processor i flips 01 in consecutive flips, and let c_j be defined as above, in relation to processor i's flip of 1. Then the probability*

that c_j is 0 for all j such that $d_j > d_i$ is at least

$$\prod_{j \mid d_j > d_i} (1 - \frac{1}{8d_j})^2.$$

Thus the probability that c_j and $next_j$ are both 0 for all j such that $d_j > d_i$ is at least

$$\prod_{j \mid d_j > d_i} (1 - \frac{1}{8d_j})^3.$$

Proof. Let f_j be defined as above. Define $f(t_0)$ to be the maximum, over all j of $f_j(t_0)$. Since $f(t_0)$ is independent of the particular execution, we can compute the time $f(t_0)$ at time t_0. Consider the following set of executions. We first consider coins that are flipped between times t_0 and $f(t_0)$. For any coin that is not a neighbor of i, or a neighbor j for which $d_j < d_i$, we flip any new coin as before: 1 with probability $1/8d_j$ and 0 otherwise. For neighbors $d_j > d_i$, if j is to flip a new coin at time t_k (which is less than $f(t_0)$ by assumption, we handle the other case below), we first check if $t_k + L_j(t_k) < f(t_0)$. If so, we just flip a regular coin, as above. If not, however, we set j's new coin to 0.

Now let E be the resulting execution to time $f(t_0)$. The claim is that at time $f(t_0)$, based on the link delay function and the coins which have already been flipped just in the partial execution E, we can now calculate t_1^E, where t_1^E is defined as the time where processor i flips again (and flips 1, by assumption). Here is how we calculate t_1^E. For each j a neighbor of i, if the value of j's coin was 0 at time $f_j(t_0)$, then set $m_j = f_j(t_0)+$ (the amount of time it takes j to send a message to i starting from time $f_j(t_0)$). If the value of j's coin was 1 at time $f_j(t_0)$, then set $m_j = f_j(t_0) + L_j(f_j(t_0))+$ (the amount of time it takes j to send a message to i starting from time $(f_j(t_0) + L_j(f_j(t_0)))$. Then, since i flips again, when it has heard from all its neighbors, and all its 1 neighbors have heard from all their neighbors, and so i can unfreeze, we have $t_1^E = \max_j m_j$. We remark that based on the link delay function, we can also calculate $f_j(t_1^E)$, for all neighbors j of i, which is the time at which i sets the coin c_j.

Now we continue the execution as follows. Now neighbors j with $d_j > d_i$ check for the first time that j flips a coin at time t_k so that $t_k + L_j(t_k) \geq f_j(t_1^E)$. Instead of flipping such a coin, j sets the new coin $= 0$.

We now show the following for neighbors j of i.

1. The executions described above occur with probability at least $\prod_{j>i}(1-(1/8d_j))^2$.

2. In an execution defined above, c_j will be 0 for all j with $d_j > d_i$.

The first claim is true, because for each j, we are deterministically setting at most two coins, one coin flipped before time $f(t_0)$ and one coin flipped after. We only set one coin after $f(t_0)$ by definition, we set the first coin such that $t_k + L_j(t_k) \geq f_j(t_1^E)$. Before t_0, we set a coin 0 if $t_k + L_j(t_k) > f(t_0)$. We claim that the next coin j flips must be flipped after time $f(t_0)$. Since if j had flipped a 1, he would flip again in time $t_k + L_j(t_k)$, and he cannot flip again any sooner with a 0, by Lemma 5.4.3.

To see the second claim, we work backward. If the coin c_j was flipped at time t_k after time $f(t_0)$, then it must be the case that $t_k + L_j(t_k) \geq f_j(t_1^E)$, otherwise j would flip again before time $f_j(t_1^E)$ and a later coin would be c_j. Also c_j must be the first coin for which $t_k + L_j(t_k) \geq f_j(t_1^E)$, because if there was an earlier coin flipped at time t_y for which $t_y + L_j(t_y) \geq f_j(t_1^E)$, then that coin would still be current at time $f_j(t_1^E)$ if it was a 1, by definition, and also if it was a 0, by Lemma 5.4.3. Thus we have set $c_j = 0$. If the coin c_j was flipped before time $f(t_0)$ but after time t_0, then the coin that was current is the coin that was current after time t_0, which we have set to 0, by design. Finally, if j does not flip a coin at all between times t_0 and t_1^E then c_j is the coin current at time t_0. Can this be a 1? No, because then j would have to flip again before time t_1 in any execution, because such a 1 would be current at time $f_j(t_0)$ and thus freeze i.

Finally, $next_j$ is the next coin processor j flips after c_j, an independent cointoss which is 0 with probability $(1 - 1/8d_j)$ for each j. □

We now show, with an analysis similar to [51], using Lemma 5.4.5 that for a constant fraction of the nodes i, we expect that some neighbor k of i flips 01, while all higher degree neighbors j of k flip c_j, and $next_j$ all 0. Then Lemma 5.4.4 guarantees that k will enter the MIS at time $t + 2\nu_j$ *entirely independent of variation in link delays.* Lemma 5.4.2 showed that a node k will always reflip in time $5\nu_j$, again entirely independent of variation in link delays. Thus independent of the link delay function, a constant fraction of the nodes will enter the MIS in time 7ν (5ν time to re-flip, and then a delay of 2ν to get ACKS from all neighbors.)

For each node k, let its degree be denoted $d(k)$, and the set of its neighbors is called $N(k)$. Let E_k be the event that processor i flips a 1 at time t, and E_k^- denote the event that k's previous coin was a 0. Let NI be the set of nodes who have at least one neighbor placed in the

MIS.

Let i be a particular node, and rename its neighbors without loss of generality $1, \ldots, d_i$ where if $j > k$, then $d(j) \geq d(k)$. We will look at the event that i flipped a 1.

Let E'_{d_i} be the event $E_{d_i} \cap E^-_{d_i}$, and for $2 \leq j \leq d_i$ let

$$E'_j = \left(\bigcap_{k=j+1}^{d_i} \neg E_k \right) \bigcap E_j \bigcap E^-_j$$

$$A_j = \bigcap_{v \in N(j), d(v) \geq d(j)} ((c_j = 0) \cap (next_j = 0)).$$

Since by Lemma 5.4.4 if A_j and E_j and occur, then j will enter the MIS, we have:

$$Pr[i \in NI] \geq \sum_{j=1}^{d_i} P[E'_j] \cdot P[A_j \mid E'_j].$$

Now the analysis proceeds as in [51]. By Lemma 5.4.5 we have

$$\begin{aligned} P[A_j \mid E'_j] &\geq \bigcap_{v \in N(j), d(v) \geq d(j)} (1 - p_j)^3 \\ &\geq \bigcap_{v \in N(j), d(v) \geq d(j)} 1 - 4p_v \\ &\geq 1 - \sum_{v \in N(j), d(v) \geq d(j)} 4p_v \\ &\geq 1/2. \end{aligned}$$

The first term can now be analyzed in a similar fashion to Luby's protocol: we are asking the probability that a neighbor, on his current coin flip, flipped a 1. We have:

$$\sum_{j=1}^{d_i} P[E'_j] \geq \frac{7}{8} P\left[\bigcup_{j=1}^{d_i} E_j\right].$$

By the technical lemma in [51]. if we let $\alpha = \sum_{j=1}^{d_i} p_j$, we then have

$$P\left[\bigcup_{j=1}^{d_i} E_j\right] \geq \frac{1}{2} \cdot min\{\alpha, 1\}$$

and thus

$$P[i \in NI] \geq \frac{7}{32} \cdot min\{\alpha, 1\}.$$

Summing over all i in V', the set of nodes who are not yet in the MIS, and do not yet have at least one neighbor in the MIS at time t, gives that a constant fraction of the remaining nodes expect to enter the MIS by time $t + 7\nu$. Notice also, from lemmas 5.4.2 and 5.4.4, that when a node j will enter the MIS is only dependent on ν_j, it local link delay to distance 2 in the graph. We thus have proved the following theorem.

Theorem 5.4.6 *The asynchronous protocol is 2-wait-free, and produces an MIS in $O(\nu \log n)$ expected time.*

5.5 Dining philosophers

5.5.1 History and definitions

The dining philosophers problem, and its extensions, model resource allocation in a distributed system. We consider the generalized dining philosophers problem, as modeled in [22]. The dining philosophers problem, studied by Dijkstra and others [33, 53, 58, 27, 52, 29], models a set of resources (such as printers, disk drives) which can only be used by one competing process at a time. The situation can be represented by a graph on the set of processors called the *conflict graph*, with an edge between two nodes if they share some resource. In this formulation, which is the standard one, we are assuming that the *conflict graph* is a subgraph of the communication graph. The intuition behind the notion of a *conflict graph* is that if two jobs are local to the same disk, for example, then they are also local to each other, so they can communicate. Each processor may handle a sequence of jobs, but tries to schedule only one job at a time; each job has a resource requirement which is a subset of the resources accessible to that processor. For a job to be executed, all of the required resources must be available for exclusive use by its processor.

We are interested in bounding the *response time* of a job. The response time is the time between when a job is assigned to a processor, and when it is executed. We will modify the MIS protocol of the previous sections to construct a dining philosophers schedule with optimal expected response time of $O(\delta_j)$ for job j, where δ_j is the number of jobs that compete with j.

Compare this to the bound of $O(\delta_j^2)$ achieved by [29] (which is still, by the way, the best known deterministic response time algorithm).

We point out as an aside that all the above bounds are stated under the normalizing assumption that job execution time, and the maximum link delay, or time it takes for a message sent by one processor to be received by its neighbor, are both less than 1 unit of time. If μ is an upper bound on the maximum length of time it takes a job to complete execution and ν is an upper bound on the maximum *link delay*, then our protocol has more precisely expected response time less than $O(\delta_j\mu + \delta_j\nu)$, for job j. (And the lower bound is similarly $\Omega(\delta_j(\mu+\nu))$). In fact, we get the *2-wait-free* property and ν and μ can be replaced by the local delays ν_1 and μ_1.

5.5.2 The reduction from asynchronous MIS

Using our asynchronous protocol, we get the optimal expected *response time* for dining philosophers. The protocol is quite simple and works as follows. Each process with no neighbor in the MIS and with an unscheduled job, runs the MIS protocol of Section 5.3. When a process enters the MIS, it first sets its ID to ∞, so that none of its neighbors can enter the MIS while its job is running. The job is then scheduled. When the job is finished executing, it sends a done message to all its neighbors, which then remove that job from their neighbor set.

We now analyze our algorithm. We note that *safety* and *self-stablization* follows from the corresponding proofs of MIS; we now prove that a job j will execute in expected $O(\delta_j)$ time.

Theorem 5.5.1 *Let $\nu(k)$ be an upper bound on the link delay of the neighbors of job k, and let ν_j be the max. over all neighbors k of job j of $\nu(k)$. Let μ_j be an upper bound on the amount of time it takes to execute any of j's neighboring jobs. Then job j will execute in $O(\delta_j(\nu_j+\mu_j))$ time.*

Proof: Let E_0 denote the event in Lemma 5.4.5, that is, j flips 01, and c_i, and $next_i$ are both 0 for all neighbors i of j such that $d_i > d_j$ then,

$$\begin{aligned}
\Pr[E_0] &\geq \frac{7}{8}\frac{1}{8\delta_j}\prod_{k\in N(j),\delta_k\geq\delta_j}(1-1/8\delta_j)^3 \\
&\geq \frac{7}{16}\left(\frac{1}{8\delta_j}\right)\text{(see MIS analysis)}
\end{aligned}$$

$$\geq \frac{1}{32\delta_j}.$$

If E_0 occurs, then by Lemma 5.4.4 j will enter the MIS in $2\nu_j$ time. Thus with normalized link delays j gets scheduled in expected $32\delta_j$ time, which is $O(\delta_j)$. The link delays, by Lemmas 5.4.2 and 5.4.4, are bounded by ν_j; and the execution delay of j's neighbors is bounded by definition by μ_j; yielding the result. □

We remark that the dependence on local link delays and job execution times in Theorem 5.5.1 shows that our dining philosophers algorithm is 2-wait-free.

We remark that the protocol and analysis holds for the static case and the dynamic case considered by [22], where new jobs can be added online to the neighbor set of old jobs. The new jobs establish communication and are acknowledged by their neighbors, and they set their initial coin flips to 0. In the dynamic case the δ_j is the maximum size of j's neighbor set before j is scheduled.

Conclusion

In this thesis, we have looked at a variety of alternate structures known under the heading of network decomposition, and revisited the precursor of network decomposition, the maximal independent set problem.

We have seen how the varying forms of network decompositions can be used as an underlying data structure, key to the design of efficient, modular algorithms that exploit locality.

We have therefore focused on the efficient construction of network decomposition itself, in parallel, distributed, and sequential models of computation.

We conclude by discussing several open questions and future directions.

Locality and removing randomness

Associated with the distributed model of computation, we would like to point out a major open question. In Chapter 2, we showed in the parallel model of computation to construct a weak network decomposition *deterministically* in polylogarithmic time in the number of vertices of the graph. However, the deterministic construction requires all parallel processors to pool global information about which sets of coin-tosses in a randomized algorithm would lead to good performance in their own local neighborhoods, and take a global "vote." Therefore, this algorithm does not work in the distributed model, where one has to a pay a cost proportional to the diameter of the network to collect global information. (In the distributed model, the best deterministic algorithm appears in Chapter 3, and is much slower). Finding a $O(poly \log n)$ time deterministic distributed algorithm for this problem is open, and is an instance of a much more general problem in the theory of constructive uses of the Erdös and Spencer *probabilistic method* [35, 62]

More generally, one can ask whether methods that perform *exhaustive search* [62, 51] or

binary search [50, 25, 56]) over a small sample space, can be adapted to work locally. The problem of (χ, d)-decomposition, finding a maximal independent set in a graph, the *dining philosophers* problem, etc. are all examples of problems where we can remove randomness from the parallel algorithm if we can globally examine sample points in a probability space that consists of a small collection of individual processors' coin tosses. No one knows how to do this locally. We remark that such a construction for (χ, d)-decomposition would immediately give one for the maximal independent set problem, but the converse is not known to be true.

Locality and Randomness

The above discussion was concerned with to what extent we can remove randomness. Conversely, Chapter 5 provides evidence that randomness can be a huge help in designing local algorithms. Our very local randomized construction in chapter 5 has enabled us to handle *asynchronous* delays on parallel and distributed networks, in such a way that they are only dependent local link delays. Deterministic constructions, on the other hand, will necessitate global dependencies and long waiting chains. We ask in general what randomized distributed algorithms can be made to depend on only local link delays.

Bibliography

[1] Y. Afek, G.M. Landau, B. Schieber, and M. Yung. The power of multimedia: combining point-to-point and multiaccess networks. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 90–104, Toronto, Canada, August 1988.

[2] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 358–370, October 1987.

[3] Yehuda Afek and Eli Gafni. End-to-end communication in unreliable networks. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 131–148. ACM SIGACT and SIGOPS, ACM, 1988.

[4] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, 1991.

[5] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, pages 15–28, Italy, September 1990. Springer-Verlag (LNCS 486).

[6] R. K. Ahuja, K. Melhourn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. Technical Report 193, MIT Operations Research Center, 1988.

[7] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. of Algorithms*, 7:567–583, 1986.

[8] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all-pairs shortest path problem. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[9] B. Awerbuch and D. Peleg. Routing with polynomial communication-space trade-off. *SIAM J. Disc. Math*, 5(2):151–162, 1992.

[10] Baruch Awerbuch. Complexity of network synchronization. *J. of the ACM*, 32(4):804–823, October 1985.

[11] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Fast distributed network decomposition. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, August 1992.

[12] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Low-diamter graph decomposition is in NC. In *Proc. 3'rd Scandinavian Workshop on Algorithm Theory*, July 1992.

[13] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near linear cost network decompositions in sequential and distributed environments. In *Proc. 35th IEEE Symp. on Foundations of Computer Science*, November 1993. To appear.

[14] Baruch Awerbuch, Lenore Cowen, and Mark Smith. Wait-free self-stabilizing symmetry breaking in expected logarithmic time. Unpublished manuscript, May 1993.

[15] Baruch Awerbuch, Andrew Goldberg, Michael Luby, and Serge Plotkin. Network decomposition and locality in distributed computation. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, May 1989.

[16] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time-optimal self-stabilizing synchronization. to appear, May 1993.

[17] Baruch Awerbuch, Yishay Mansour, and Nir Shavit. End-to-end communication with polynomial overhead. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989.

[18] Baruch Awerbuch and Silvio Micali. Dynamic deadlock resolution protocols. In *Proc. 27th IEEE Symp. on Foundations of Computer Science*, October 1986.

[19] Baruch Awerbuch, Boaz Patt, David Peleg, and Mike Saks. Adapting to asynchronous dynamic networks with polylogarithmic overhead. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 557–570, 1992.

[20] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 268–277, October 1991.

[21] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, 1990.

[22] Baruch Awerbuch and Mike Saks. A dining philosophers algorithm with polynomial response time. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990.

[23] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[24] Bonnie Berger and Lenore Cowen. Fast deterministic constructions of low-diameter network decompositions. Technical Report MIT/LCS/TR-460, Lab. for Computer Science, MIT, Cambridge, MA, April 1991.

[25] Bonnie Berger and John Rompel. Simulating $(\log^c n)$-wise independence in NC. *J. of the ACM*, 38(4):1026–1046, October 1991. Preliminary version appeared in FOCS '89.

[26] P. A. Bloniarz. A shortest path algorithm with expected time $o(n^2 \log n \log^* n)$. Technical Report 80-3, Dept. of Computer Science, State University of Albany, August 1980.

[27] K. Chandy and J. Misra. The drinking philosophers problem. *ACM TOPLAS*, 6(4):632–646, October 1984.

[28] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comput. Syst.*, 3(1):63–75, February 1985.

[29] M. Choy and A.K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proc. 24th ACM Symp. on Theory of Computing*, 1992.

[30] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. In *Proc. 35th IEEE Symp. on Foundations of Computer Science*, 1993. To appear.

[31] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[32] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643–644, 1974.

[33] E.W. Dijkstra. Hierarchical ordering of sequential processes. *ACTA Informatica*, pages 115–138, 1971.

[34] E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *CACM*, 17:643–644, Nov 1974.

[35] P. Erdös and J. Selfridge. On a combinatorial game. *Journal of Combinatorial Theory Series A*, 14:298–301, 1973.

[36] M. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. of the ACM*, pages 596–615, 1987.

[37] A. M. Frieze and G. R. Grimmet. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10, 1985.

[38] H. Gabow. Scaling algorithms for network problems. *J. Comput. and Syst. Sci.*, 31:148–168, 1985.

[39] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Lang. and Syst.*, 5(1):66–77, January 1983.

[40] M. R. Garey and D. S. Johnson. *Computers and Intractability: a guide to the theory of NP-completeness.* Freeman, San Francisco, CA, 1981.

[41] H. Jakobsson. Mixed-approach algorithms for transitive closure. In *Proceedings of the 10th ACM Symposium on Principles of Database Systems*, 1991.

[42] D. Johnson. Efficient algs. for shortest paths in sparse networks. *J. of the ACM*, pages 1–13, 1977.

[43] D. R. Karger, D. Koller, and S. J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[44] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *J. of the ACM*, 32(4):762–773, October 1985.

[45] P. N. Klein. A parallel randomized approximation scheme for shortest paths. Technical Report CS-91-56, Brown University, August 1991.

[46] Chengdian Lin and Janos Simon. Observing self-stabilization. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, August 1992. to appear.

[47] N. Linial and M. Saks. Decomposing graphs into regions of small diameter. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 320–330. ACM/SIAM, January 1991.

[48] Nathan Linial. Locality as an obstacle to distributed computing. In 27th *Annual Symposium on Foundations of Computer Science*. IEEE, October 1987.

[49] M. Luby. A simple parallel algorithm for the maximal independent set problem *SIAM J. on Comput.*, 15(4):1036–1053, November 1986.

[50] M. Luby. Removing randomness in parallel computation without a processor penalty. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 162–173. IEEE, October 1988.

[51] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. on Comput.*, 15(4):1036–1053, November 1986.

[52] J. Lundelius and N. Lynch. Synthesis of efficient drinking philosophers algorithms. unpublished manuscript, January 1988.

[53] N. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal Of Computation And Systems Sciences*, 23(2):254–278, October 1981.

[54] Alain Mayer, Yoram Ofek, Rafail Ostrovsky, and Moti Yung. Self-stabilizing symmetry breaking in constant-space. In *Proc. 24th ACM Symp. on Theory of Computing*, 1992.

[55] C. C. McGeoch. A new all-pairs shortest-path algorithm. Technical Report 91-30, DIMACS, 1991.

[56] R. Motwani, J. Naor, and M. Naor. The probabilistic method yields deterministic parallel algorithms. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 8–13. IEEE, October 1989.

[57] Alessandro Panconesi and Aravind Srinivasan. Improved algorithms for network decompositions. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 581–592, 1992.

[58] M. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th POPL*, pages 133–138, 1981.

[59] Satish Rao. Finding small edge cuts in planar graphs. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 229–240, 1992.

[60] R. Seidel. On the all-pairs-shortest-path problem. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 745–749, May 1992.

[61] M.A.S. Smith. *Fast Wait-Free Symmetry Breaking in Distributed Systems.* PhD thesis, Department of EECS, MIT, 1993.

[62] J. Spencer. *Ten Lectures on the Probabilistic Method.* SIAM, Philadelphia, PA, 1987.

[63] P. M. Spira. A new algorithm for finding all shortest paths in a graph of positive arcs in average time $o(n^2 \log^2 n)$. *SIAM J. on Comput.*, 2, 1973.

[64] Eugene Styer and Gary Peterson. Improved algorithms for distributed resource allocation. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 105–116. ACM SIGACT and SIGOPS, ACM, 1988.

[65] J. D. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. *SIAM J. on Comput.*, 20, 1991.