MIT/LCS/TM-460

FAST DETERMINISTIC CONSTRUCTIONS OF LOW-DIAMETER NETWORK DECOMPOSITIONS

Bonnie Berger
Lenore Cowen

December 1991

Fast Deterministic Constructions of Low-Diameter Network Decompositions

Bonnie Berger[*]
Lenore Cowen[†]

Laboratory for Computer Science
and
Department of Applied Mathematics
Massachusetts Institute of Technology
Cambridge, MA 02139

April 3, 1991

Abstract

Finding a low-diameter decomposition of a network is one of the central problems in the theory of distributed systems. Many researchers have studied this problem in the hopes of obtaining improved distributed algorithms for maximal independent set, (Δ+1)-coloring, depth first search, all-pairs shortest paths, local routing, and distributed directories. We present here the fastest known deterministic distributed algorithm for the *optimal* decomposition of a network.

More formally, we are interested in getting a decomposition of a graph into *clusters* where the distance between any two nodes in the same cluster is at most d, and the clusters are colored with χ colors, such that the distance between clusters of the same color is strictly greater than 1. We give a deterministic algorithm for the optimal $\chi = O(\log n)$, $d = O(\log n)$ decomposition which runs in $O(n^{\epsilon})$ time, for any $\epsilon > 0$, on an n node network in the asynchronous distributed model. It is important to get a decomposition which is logarithmic in both χ and d to implement the applications efficiently. The previous best known deterministic algorithm for this problem took linear time [7, 10].

In addition, we look at finding low-diameter decompositions in the PRAM model. We describe the first deterministic *NC* algorithm for low-diameter decomposition. It finds a $\chi = O(\log^2 n)$, $d = O(\log n)$ decomposition in $O(\log^5(n))$ time using $O(n^2)$ processors. Previously no deterministic *NC* algorithm for network decomposition was known.

[*]Supported by an NSF Postdoctoral Research Fellowship.

[†]Supported in part by DARPA contracts N00014-87-K-0825 and N00014-89-J-1988, Air Force Contract OSR-89-02171, Army Contract DAAL-03-86-K-0171 and Navy-ONR Contract N00014-19-J-1698

1 Introduction

In a distributed network, a node is assumed to communicate only with its neighboring nodes, and to have no initial information about the topology of the entire network. When we are given a network problem that depends only on local properties of the underlying graph, we solve it most efficiently if we can work separately on different local regions in the graph. This is because communication across high-diameter subgraphs is expensive.

The best existing distributed algorithms for many graph problems exploit locality by utilizing the decomposition into regions of small diameter considered in this paper. Such a decomposition serves as a distributed *data structure* which improves the performance of many algorithms which can synchronize [3], [6], route across [7], or run on the local regions [4].

It is especially important, for these types of distributed applications, that our algorithms be *deterministic.* This is because we are running graph algorithms on top of this assuming the underlying decomposition is good, and these algorithms could fail to perform well if the underlying decomposition is bad. Furthermore, it seems to be difficult even to *check* after a randomized run of an algorithm if one region far away is bad; it might take order of the diameter of the entire network to find out. This is least crucial when we are dealing with a static network, where we pay the overhead to get a good decomposition once, and then can use it over and over as a localizing data structure for running future graph algorithms on the network. If, however, we have a *dynamic network* where nodes are being added and deleted all the time, links are being added, or failing due to faults, etc., so that we have to continually update to run on a new network topology (see [8]), it becomes crucial to have good fast deterministic decompositions. We do not wish to continually pay a large overhead in time, locality, or uncertainty every time we update the topology of the network on line.

1.1 Formal Definition of the Problem

Given a graph $G = (V, E)$, a (χ, d, λ)-*decomposition* is defined to be a coloring of all the nodes of the graph with the following properties:

- there are χ colors,
- the nodes of each color are partitioned into *clusters,*
- the *weak diameter* of any *cluster* of a single color class, is at most d, (The *weak diameter* of a subgraph, H is defined to be the maximum over all pairs of nodes x and y in H, of the minimum length path between x and y *where this minimum is taken over all edges in* G, i.e. the distance is allowed to "shortcut" through nodes in other color classes.), and
- clusters of the same color are at least (weak) distance λ apart.

When $\lambda = 1$, we will abbreviate this as a (χ, d)-decomposition. When χ and d are both polylogarithmic, we refer to this as a *low-diameter* decomposition.

A (χ, d)-decomposition can be thought of as a generalization of the standard graph coloring problem, where χ is the number of colors used, and the clusters are supernodes of diameter d.

Graph decomposition problems of this type were first considered by [14, 13, 4], and later by [7, 1, 10]. Throughout this abstract we use the formulation of Awerbuch et. al. [4], and Linial-Saks [10], because of its elegant combinatorial structure; an algorithm for this type of decomposition can be transformed into an algorithm which produces an Awerbuch-Peleg *coarsening partition* (as defined in [7]), which is the useful formulation of the decomposition for many of the applications.

Awerbuch and Peleg's many applications include the construction of spanners and tree-covers for a network, routing, and online tracking of mobile users [7].

Awerbuch-Peleg [7] gave the first algorithm for $(O(\log n), O(\log n))$-decomposition, and Linial-Saks [10] further proved that this is the *optimal* decomposition, i.e. there exist graphs for which this is the best possible (χ, d)-decomposition. The Awerbuch-Peleg algorithm was a deterministic (distributed) greedy one, which runs in *linear* time. Linial-Saks [10] also gave a randomized distributed (trivially also an *RNC*) algorithm which runs in $O(\log^2 n)$ time.

1.2 Our Results

We look at *deterministic* constructions of (χ, d, λ)-decompositions. We state our results with $\lambda = 1$; however we remark that we can always generalize this to create a buffer zone between clusters as long as we set the parameters λ and d so that $d \geq \Omega(\lambda \log n)$.

Our principal result is a deterministic distributed algorithm for the optimal $(O(\log n), O(\log n))$-decomposition of an arbitrary graph in $n^{O\left(\sqrt{\log\log n}/\sqrt{\log n}\right)}$ time, which runs on an asynchronous network. (This is $O(n^\epsilon)$ time for any $\epsilon > 0$.)

The previous best known deterministic algorithm for this problem took linear time [7, 10]. Awerbuch et. al. [4] in fact match our time, but get a substantially worse decomposition; namely, a $(n^{O\left(\sqrt{\log\log n}/\sqrt{\log n}\right)}, n^{O\left(\sqrt{\log\log n}/\sqrt{\log n}\right)})$ decomposition. These larger clusters are not nearly as efficient in exploiting locality in the network.

We then turn to the PRAM model, giving the first deterministic *NC* algorithm for finding a $(O(\log^2 n), O(\log n))$-decomposition for an arbitrary graph. Our algorithm runs in $O(\log^5(n))$ time, and uses $O(n^2)$ processors. Previously, no *NC* algorithm for (χ, d)-decomposition was known.

We obtain our *NC* algorithm by derandomizing the *RNC* algorithm of Linial and Saks. The hard part is in finding a pairwise independent version of their *RNC* algorithm. To do this, we need to modify the distribution and the algorithm to be sensitive to graphs which have non-uniform densities of nodes within different local regions. Scaling techniques in previous work only took into account variations in node degree. We, on the other hand, employ a more complicated measure.

It is interesting to note that the Linial-Saks algorithm *cannot* be shown to work with constant-wise independence; one can construct graphs for which there will be no good sample point in a sample space with only constant-wise independence. It even seems doubtful that the Linial-Saks algorithm would work with polylogarithmic independence.

Saks [15] in fact posed the problem of finding an *NC* algorithm for the low-diameter decomposition problem in the context of an important step toward getting a polylogarithmic-time deterministic distributed algorithm for this problem. Notice that it is only a first step: when we derandomize an *RNC* (and randomized distributed) algorithm to get an *NC* algorithm, we do not automatically have a deterministic distributed algorithm. This is because even if we know that a good sample point exists in a small sample space, we still might not be able to figure out quickly which point this is; the benefit of a specific sample point might be a global function which we need to look over all the vertices to compute. This is no trouble in the parallel domain, where all nodes can communicate in $O(1)$ time, and we can look at a sample point over an entire graph. A polylogarithmic-time deterministic distributed algorithm for the low-diameter decomposition problem would give polylogarithmic-time deterministic distributed algorithms for maximal independent set and $(\Delta+1)$-coloring [4, 10]. It is perhaps the most important open question in the theory of distributed computing [10].

Distributed Applications

Consider the problem of all-pairs shortest paths parameterized in terms of λ; i.e. we are interested in looking at all-pairs shortest paths of distance at most λ between nodes in an asynchronous network. One can easily solve this problem with communication complexity $O(mn)$ and time $O(\lambda)$; [1] have communication complexity $O(n^2 \log n)$ and time $O(n \log n)$, where n is the number of nodes in the entire network. We achieve communication complexity $O(n^{2+\epsilon})$ and time λn^{ϵ}; thereby improving the first algorithm in terms of communication complexity, and the second in terms of time. This is useful for local routing on a distributed network and distributed directories, as well [7]. For other applications, see Awerbuch-Sipser [8], Afek-Ricklin [1], and Awerbuch-Peleg [7].

2 The Distributed Algorithm

In this section, we introduce the new distributed algorithm, **Color**, which recursively builds up a cluster structure. It calls on a procedure, **Create-New-Color**, which runs a modified version of the greedy algorithm on separate clusters. First we need to review the sequential greedy algorithm.

2.1 The Sequential Greedy Algorithm

Awerbuch-Peleg [5] present a simple greedy algorithm for getting a $(O(\log n), O(\log n))$-decomposition in linear time. The algorithm colors a constant fraction of the nodes with a single color. Pick a color. The algorithm picks an arbitrary node, (call it a *center* node) and greedily grows a ball around it of minimum radius r, such that a constant 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 $r \leq \log n$ for which this condition holds. We then put the interior of the ball into our color class, and the entire ball is removed from our graph. (The *border* (those nodes whose distance from the center node is exactly r) will not be colored with the current color). Then we pick another arbitrary node, and do the same thing, until all nodes in the graph have been processed. We then return all the uncolored nodes (the border nodes) to the graph, and begin again on a new color.

2.2 The New Distributed Algorithm

In this section, we give a new recursive deterministic distributed algorithm for obtaining a $(\log n, \log n)$-decomposition[1], which runs in $n^{O\left(\sqrt{\log\log n}/\sqrt{\log n}\right)}$ time in the asynchronous model. We note that the algorithm given can be trivially modified to get a $(\log n, t \log n, t)$-decomposition.

The algorithm is implicitly taking higher and higher powers of the graph, where 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 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 a variant of the sequential greedy algorithm, described in Subsection 2.1, to in parallel *recolor* these separate clusters without collisions.

[1]For ease of notation, we leave out the big oh in χ, d, and λ for the remainder of this paper. We note that we give the exact constants in our proofs.

In this discussion, recall that all distances between nodes, including those in the same cluster, are assumed to be *weak* distances, and the diameters of the clusters are always in terms of the weak diameter (see Section 1.1).

```
Algorithm: **Color**($G$)

**Input:** graph $G$
**Output:** a $(\log n, \log n)$-decomposition of $G$

  1. Compute $G^{2\log n}$.
  2. If $G$ has less than $x$ nodes, run the sequential greedy algorithm on $G^{2\log n}$, 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 $G^{2\log n}$ 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 $x \log n$ coloring of $G$ with separation $2\log n$)*
  7. Do sequentially, for $i = 1$ to $\log n$, **Create-New-Color**($G$, $i$)
     *(this gives a $\log n$ coloring of $G$ with separation 1)*
```

We now present the procedure **Create-New-Color**.

```
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 $(x\log n, 4\log^2 n, 2\log n)$-
decomposition on the old-colored nodes of $G$ and a $(i-1, 2\log n, 1)$-decomposition on the new-
colored nodes of $G$
**Output:** graph $G$ with new and old colored nodes such that there is a $(x\log n, 4\log^2 n, 2\log n)$-
decomposition on the old-colored nodes of $G$ and a $(i, 2\log n, 1)$-decomposition on the new-colored
nodes of $G$

  1. Do sequentially, for $j = 1$ to $x \log n$,
     "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 of all old-colored nodes within $\log n$ distance from
             its cluster.
           • The leader runs a variant of the simple greedy algorithm as follows: the *centers* of
             the balls grown out are always picked (arbitrarily) from the nodes in the old-color
             $j$ cluster; but the interiors and borders of the balls which are then claimed, include
             any of the old-colored nodes (not just those of color $j$) within the ball. As before,
             the leader colors the interior of the balls with new color $i$, and considers the entire
             ball removed from the graph. The leader continues until all nodes of old-color $j$ in
             the graph have been processed.
```

2. Re-introduce all removed nodes.

Lemma 2.1 *If $x = 2^{\sqrt{\log n}\sqrt{\log\log n}}$, the running time of* **Color** *is $n^{O\left(\sqrt{\log\log n}/\sqrt{\log n}\right)}$* .

Proof The running time blows up in recursive levels. In particular, the factor of blowup is $(2\log n)^L$, for the Lth level. Since L is at most $(\log n)/(\log x)$, the maximum blowup we get is $(2\log n)^{(\log n)/(\log x)}$. Since we use $x\log^3 n$ neighbor to neighbor time to communicate in the current graph we're dealing with, we get a running time of at most $(x\log^3 n)(2\log n)^{(\log n)/(\log x)}$. If we choose $x = 2^{\sqrt{\log n}\sqrt{\log\log n}}$, we get the running time bounded by $2^{O(\sqrt{\log n}\sqrt{\log\log n})}$ which is exactly $n^{O\left(\sqrt{\log\log n}/\sqrt{\log n}\right)}$. □

Theorem 2.2 *There is a deterministic distributed asynchronous algorithm which given a graph $G = (V, E)$, finds a $(\log n, \log n)$-decomposition in $n^{O\left(\sqrt{\log\log n}/\sqrt{\log n}\right)}$ time.*

3 Decomposing a Graph into Regions of Low-Diameter is in *NC*

Here we present the first deterministic *NC* algorithm for finding a low-diameter decomposition problem. We achieve this by modifying an RNC algorithm of Linial-Saks to depend only on pairwise independence, and then remove the randomness. To get our newly-devised pairwise independent benefit function to work, we had to employ a non-trivial scaling technique. Scaling has been used previously only on the simple measure of node degree in a graph.

3.1 The *RNC* Algorithm of Linial-Saks

To simulate the greedy algorithm randomly, Linial-Saks [10] still consider each of the $O(\log n)$ 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 weak 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. In their algorithm, 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; where we explain how they set B, below). It is assumed that each node has a unique ID associated with it. 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 the abstract $d(z,y)$ will always denote the *weak* distance between a z and y. 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 bounds the diameter of the resulting clusters by $2B$. They then show they can set $B = O(\log n)$ so that 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 [10] for a discussion of trade-offs between diameter and number of colors; in [10], 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 there will be no good sample point in a sample space with only constant-wise independence. 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.

3.2 A Pairwise Independent *RNC* Algorithm

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

3.2.1 The Algorithm

The algorithm will use $O(\log^2 n)$ colors: one for each of the $O(\log n)$ phases of $O(\log n)$ iterations each. At the end of each iteration, nodes assigned the corresponding color are eliminated from the graph. We are able to show that our algorithm colors a constant fraction of the remaining nodes in each phase. The phases are designed to mimic the *RNC* algorithm of Linial and Saks [10]. We need the more complicated iterations inside each phase to be able to get the algorithm to work with pairwise independence. We need to change the distribution over each iteration to capture nodes in successively sparser regions of the graph. Our last iteration in each phase is identical to running one phase of Linial and Saks, on the remaining graph that has not yet been colored by our algorithm.

Define $T_y = \sum_{z|d(z,y)<B} p^{d(z,y)}$, and $\Delta = \max_{\forall y \in G} T_y$. Each phase will have $O(\log n)$ iterations, wherein each iteration, i, colors a constant fraction of the nodes y with T_y between $\Delta/2^{i+1}$ and $\Delta/2^i$. Note that T_y decreases from iteration to iteration, but Δ is fixed.

The algorithm for each iteration i of a phase is as follows. Let $x = 2^i/(5\Delta)$. First each node y selects an integer radius r_y pairwise independently at random according to the distribution that follows. It is a *scaled* truncated geometric distribution (as opposed to the simple truncated geometric distribution of Linial-Saks), which is defined in terms of three parameters, p, B, and x. Since x is increased throughout a phase, and then reset at the beginning of the next phase, our new distribution varies dynamically. For $y = 1, \ldots, n$:

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

We can assume every node has a unique ID [10]. 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 doesn't broadcast.) At the end of the iteration, all the nodes colored are removed from the graph.

3.2.2 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 [10] 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:

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

We note that the Linial and Saks algorithm cannot be shown to work with this lower bound.

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

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

It will be convenient for us to define the *benefit* [9, 12] of a sample point $R = <r_1, \ldots, r_n>$ for a single node y, as

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

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

Recall that $T_y = \sum_{z \mid d(z,y) < B} p^{d(z,y)}$.

Lemma 3.1 *If $p \leq 1/2$, then $E[\mathrm{B}_y(R)] \geq pxT_y - px^2(T_y^2/2)$.*

Proof Since $E[X_{y,z}] = xp^{d(z,y)}$, and $E[Z_{y,z}] = xp^{d(z,y)+1}$, we can rewrite

$$E[B_y(R)] = pxT_y - px^2 \left(\sum_{u > z \mid \substack{d(z,y) < B \\ d(u,y) \leq B}} p^{d(z,y)+d(u,y)} \right)$$

It thus remains to show that

$$\frac{T_y^2}{2} \geq \sum_{u > z \mid \substack{d(z,y) < B \\ d(u,y) \leq B}} p^{d(z,y)+d(u,y)} \qquad (1)$$

We proceed as follows:

$$\begin{aligned}
\frac{T_y^2}{2} &= \frac{1}{2} \left[\sum_{z \mid d(z,y) < B} p^{2d(z,y)} + \left(2 \sum_{z_1 < z_2 \mid \substack{d(z_1,y) < B \\ d(z_2,y) < B}} p^{d(z_1,y)+d(z_2,y)} \right) \right] \\
&= \frac{1}{2} \sum_{z \mid d(z,y) < B} p^{2d(z,y)} + \sum_{u > z \mid \substack{d(z,y) < B \\ d(u,y) \leq B}} p^{d(z,y)+d(u,y)} - \sum_{u > z \mid \substack{d(z,y) < B \\ d(u,y) = B}} p^{d(z,y)+B}
\end{aligned}$$

Since the middle term is exactly the right hand side of inequality 1, it remains only to show that the difference of the first and last terms is positive. We have, if $d(z,y) < B$, then $2d(z,y) \leq d(z,y) + B - 1$. Therefore

$$\begin{aligned}
\frac{1}{2} \sum_{z \mid d(z,y) < B} p^{2d(z,y)} &\geq \frac{1}{2} \sum_{z \mid d(z,y) < B} p^{d(z,y)+B-1} \\
&= \frac{1}{2p} \sum_{z \mid d(z,y) < B} p^{d(z,y)+B}
\end{aligned}$$

and for $p \leq 1/2$, since conditioning a sum further can only decrease the number of terms, we get

$$\frac{1}{2} \sum_{z|d(z,y) < B} p^{2d(z,y)} \geq \sum_{\substack{u|d(u,y) = B, \\ z|d(z,y) < B}} p^{d(z,y)+B}$$

□

We define the set D_i at the ith iteration of a phase as follows:

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

It is clear from the definition that each y can be in at most one set D_i, in the proof of Lemma 3.3 we will argue that every y falls into D_i for some iteration i as well. The sets D_i, dynamically constructed over the iterations of a phase, therefore form a partition of the nodes of the graph.

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

$$B_{\mathcal{I}}(R) = \sum_{D_i} B_y(R). \tag{2}$$

Recall that $\Delta = \max_{\forall y \in G} T_y$, and at the ith iteration of a phase, $x = 2^i/(5\Delta)$. 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 3.2 *In the ith iteration, we expect to color $2p/25$ of those $y \in D_i$.*

Proof

$$\begin{aligned} E[\text{ \# of } y \in S|D_i] &= \sum_{y \in D_i} Pr[y \in S] \\ &\geq E[B_{\mathcal{I}}(R)] \\ &\geq \sum_{y \in D_i} p\frac{2^i}{5\Delta}T_y - p\left(\frac{2^{2i}}{25\Delta^2}\right)\frac{T_y^2}{2} \end{aligned}$$

by Lemma 3.1. Since we want a lower bound, we substitute $T_y \leq \frac{\Delta}{2^i}$ in the positive term and $T_y \geq \frac{\Delta}{2^{i+1}}$ in the negative term, giving

$$\begin{aligned} E[\text{ \# of } y \in S \mid y \in D_i] &\geq \sum_{y \in D_i} p\frac{1}{10} - p\frac{1}{50} \\ &= \frac{2}{25}p|D_i| \end{aligned}$$

□

The next lemma gives us that the expected number of phases is $O((\log n)/(\log(2p/25))) = O(\log n)$.

Lemma 3.3 *Suppose $V' \subseteq V$ is the set of nodes present in the graph at the beginning of a phase. After $\log(5\Delta)$ iterations of a phase, the expected number of nodes colored is $(2p/25)|V'|$.*

Proof Since for all y, $T_y \geq 1$, over all iterations, and since $x \to 1$, then there must exist an iteration where $xT_y \geq 1/10$. Since T_y can not increase (it can only decrease if we color and remove nodes in previous iterations), and $xT_y \leq 1/5$ in the first iteration for all y, we know that for each y there exists an iteration in which $1/5 \geq xT_y \geq 1/10$. 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 3.2, we expect to color $2p/25$ 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 $(2p/25)$ fraction overall.
□

It remains to set B so that we can determine the maximum weak diameter of our clusters, which also influences the running time of the algorithm. By Lemma 3.3, we have that the probability of a node being colored in a phase is $2p/25$. Thus, the probability that there is some node which has not been assigned a color in the first l phases is at most $n(1-(2p/25))^l$. By selecting B to be $\frac{25\log n+\omega(1)}{2p}$, it is easily verified that this quantity is $o(1)$.

Theorem 3.4 *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.*

Remark: The distribution used by the *RNC* algorithm can be modified so that the algorithm finds a $(\log^2 n, t\log n, t)$-decomposition, for any integer t.

3.2.3 The Pairwise Independent Distribution

We have shown that we expect our *RNC* algorithm 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 $i=1,\ldots,n$:

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

Without loss of generality, let p and x be powers of 2. Let $r = B\log(1/p)+\log(1/x)$. Note that since $B = O(\log n)$, we have that $r = O(\log n)$. In order to construct the sample space, we choose $W \in \mathbf{Z}_2^l$, 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 that its kth 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

$$\begin{aligned} r_i &= NIL && \text{if } t \in [1,..,\log(1/x)] \\ &= j && \text{if } t \in (\log(1/x)+j\log(1/p),..,\log(1/x)+(j+1)\log(1/p)], \text{ for } 0 \le j \le B-1 \\ &= B && \text{otherwise.} \end{aligned}$$

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 [9, 12] 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.

3.3 Our *NC* Algorithm

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 our 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 [9, 11, 2]). 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 [12] 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.

$$B_{\mathcal{I}}(R) = \sum_{y \in D_i} B_y(R) = \sum_{y \in D_i} \left(\sum_{z | d(z,y) < B} Z_{y,z} - \sum_{u > z | \substack{d(z,y) < B \\ d(u,y) \leq B}} Z_{y,z} X_{y,u} \right), \tag{3}$$

where recall $D_i = \{y | \Delta/2^{i+1} \leq T_y \leq \Delta/2^i \wedge (y \notin D_h \text{ for all } h < i)\}$. The binary search is over the bits of W (see Section 3.2.3): 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. 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, the algorithm will perform within at least a multiplicative factor of p of its performance in Lemmas 3.2 and 3.3, 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 3.2.3. Notice that our indicator variables have the following meaning:

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

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

$$E[\sum_{(z,u) \in S} X_{y,z} X_{y,u} \mid \omega_1^{(1)} = b_{11}, \omega_2^{(1)} = b_{12}, \ldots, \omega_t^{(q)} = b_{qt}], \tag{4}$$

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

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 4 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,

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

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 $S_\alpha = \{z \in S' \mid z_{t+1} \cdots z_{\log n} = \alpha\}$. We further partition each S_α into $S_{\alpha,0} = \{z \in S_\alpha \mid \sum_{i=1}^{t} z_i \omega_i^{(q)} = 0 \pmod 2\}$ and $S_{\alpha,1} = S_\alpha - S_{\alpha,0}$. 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 $z \in S_{\alpha,j}$, and $u \in S_{\alpha,j'}$, then $Y_{z,q} = Y_{u,q}$ iff $j = j'$, and
3. if $z \in S_\alpha$ and $z' \in S_{\alpha'}$, where $\alpha \neq \alpha'$, then $Pr[Y_{z,q} = Y_{u,q}] = Pr[Y_{z,q} \neq Y_{u,q}] = 1/2$.

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

$$
\begin{aligned}
& E[\sum_{(z,u)\in S'} X_{y,z}X_{y,u}] \\
&= E[\sum_{(z,u)\in S'} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q}] \\
&= E[\sum_{\alpha}\sum_{(z,u)\in S_\alpha} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q} + \sum_{(\alpha,\alpha')}\sum_{z\in S_\alpha}\sum_{u\in S_{\alpha'}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q}] \\
&= \sum_{\alpha} E[\sum_{(z,u)\in S_{\alpha,0}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q} + \sum_{(z,u)\in S_{\alpha,1}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q} + 2\sum_{z\in S_{\alpha,0}}\sum_{u\in S_{\alpha,1}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q}] \\
&\quad + \sum_{(\alpha,\alpha')} E[\sum_{z\in S_\alpha}\sum_{u\in S_{\alpha'}} \gamma(z,y)\gamma(u,y)Y_{z,q}Y_{u,q}] \\
&= \sum_{\alpha}\left[\frac{1}{2}\sum_{(z,u)\in S_{\alpha,0}} \gamma(z,y)\gamma(u,y) + \frac{1}{2}\sum_{(z,u)\in S_{\alpha,1}} \gamma(z,y)\gamma(u,y) + 0\right] + \sum_{(\alpha,\alpha')}\frac{1}{4}\left(\sum_{z\in S_\alpha}\gamma(z,y)\right)\left(\sum_{u\in S_{\alpha'}}\gamma(u,y)\right) \\
&= \frac{1}{2}\sum_{\alpha}\left[\left(\sum_{z\in S_{\alpha,0}}\gamma(z,y)\right)^2 - \sum_{z\in S_{\alpha,0}}\gamma(z,y)^2 + \left(\sum_{z\in S_{\alpha,1}}\gamma(z,y)\right)^2 - \sum_{z\in S_{\alpha,1}}\gamma(z,y)^2\right] \\
&\quad + \frac{1}{4}\left[\left(\sum_{\alpha}\sum_{z\in S_\alpha}\gamma(z,y)\right)^2 - \sum_{\alpha}\left(\sum_{z\in S_\alpha}\gamma(z,y)\right)^2\right].
\end{aligned}
$$

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 3.3 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 3.5 *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.*

4 Acknowledgments

Many thanks to Baruch Awerbuch, Tom Leighton, David Peleg, and John Rompel for helpful discussions. In particular, we would like to thank Baruch Awerbuch and David Peleg for all their help clarifying issues in the distributed model, and for suggesting distributed applications.

References

[1] Y. Afek and M. Ricklin. Sparser: A paradigm for running distributed algorithms. Unpublished manuscript, April 1990.

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

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

[4] 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*, November 1989.

[5] Baruch Awerbuch and David Peleg. Routing with polynomial communication-space trade-off. Technical Memo TM-411, MIT, Lab. for Computer Science, September 1989.

[6] Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990.

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

[8] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 206–220, October 1988.

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

[10] N. Linial and M. Saks. Decomposing graphs into regions of small diameter. In *Proceedings of the Second Annual ACM/SIAM Symposium on Discrete Algorithms*, pages 320–330. ACM/SIAM, January 1991.

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

[12] M. Luby. Removing randomness in parallel computation without a processor penalty. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 162–173. IEEE, October 1988.

[13] D. Peleg. Complexity considerations for distributed data structures. Technical Report CS89-31, The Weizmann Institute, December 1989.

[14] D. Peleg. Sparse graph partitions. Technical Report CS89-01, The Weizmann Institute, February 1989.

[15] M. Saks. Decomposing graphs into regions of small diameter, January 1991. Talk at ACM/SIAM Symposium on Discrete Algorithms.