

Network Decomposition and Locality in Distributed Computation (Extended Abstract)

*Baruch Awerbuch**

Department of Mathematics and
Laboratory for Computer Science
M.I.T.
Cambridge, MA 02139

Andrew V. Goldberg†

Department of Computer Science
Stanford University
Stanford, CA 94305

Michael Luby‡

International Computer Science Institute
Berkeley, CA 94704

Serge A. Plotkin§

Department of Computer Science
Stanford University
Stanford, CA 94305

May 1989

Abstract

We introduce a concept of *network decomposition*, the essence of which is to partition an arbitrary graph into small-diameter connected components, such that the graph created by contracting each component into a single node has low chromatic number. We present an efficient distributed algorithm for constructing such a decomposition, and demonstrate its use for design of efficient distributed algorithms.

Our method yields new deterministic distributed algorithms for finding a maximal independent set and for $(\Delta + 1)$ -coloring of graphs with maximum degree Δ . These algorithms run in $O(n^\epsilon)$ time for any $\epsilon > 0$, while the best previously known deterministic algorithms required $\Omega(n)$ time. Our techniques can be used to remove randomness from the previously known most efficient distributed Breadth-First Search algorithm, without increasing the asymptotic communication and time complexity.

*Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

†Research partially supported by NSF Presidential Young Investigator Grant CCR-8858097, IBM Faculty Development Award, and ONR Contract N00014-88-K-0166.

‡On leave of absence from the Computer Science Department, University of Toronto, research partially supported by NSERC of Canada operating grant A8092.

§Research partially supported by ONR Contract N00014-88-K-0166.

1 Introduction

A distributed network of processors is described by an undirected graph $G = (V, E)$, where $|V| = n$. There is a processor located at each node of G that can communicate directly only with processors at neighboring nodes, *i.e.*, all communication is *local*. As a result, locality is a very important issue in distributed computation, and algorithms that use only local communication are of special interest. Parallel deterministic algorithms rely heavily on global communication, since in parallel models of computation it is easy to collect information from all the nodes quickly. In the distributed network this is not the case, because the communication channels are part of the problem input. Using only communication links of the network, it takes at least the diameter of G time steps to gather global information. In general, the diameter of G could be as large as n . Therefore some problems, known to be in NC, are not known to be solvable in polylogarithmic time in a distributed system. An representative example of such a problem is the *Maximal Independent Set* (MIS) problem. Linial [12] was the first to use sophisticated graph-theoretic techniques to study locality in distributed systems and to point out the importance of the MIS problem in this context.

We say that a function can be computed *locally* if it can be computed deterministically in time that is significantly smaller than n , for any possible diameter of the network. For some problems, such as the MIS problem, global communication can be avoided if the interconnection network has a small degree (see e.g. [7, 8]). A natural approach to extend this idea to a high degree network is to attempt to decompose the network into clusters of small diameter, so that the graph induced by clusters (the *cluster graph*) has a small degree. Given such a decomposition, global communication within a cluster is relatively inexpensive (because of its small diameter), and global communication between clusters can be avoided. Unfortunately, it is easy to construct a graph where, even if we allow cluster diameter to be as high as \sqrt{n} , the degree of the cluster graph is $\Omega(\sqrt{n})$ [Linial and Saks, personal communication]. We can overcome this problem by showing that in order to avoid global communication between clusters, it is sufficient to use decomposition with a relaxed requirement. Instead of requiring the cluster graph to have a small degree, we require it to have a small arboricity¹, and therefore a small chromatic number.

This motivates the main technique of this paper, which appears to be very useful for design of local distributed algorithms. We introduce the concept of a $(d(n), c(n))$ -*decomposition*, which is interesting from the graph-theoretic point of view as well. A $(d(n), c(n))$ -decomposition of a graph $G = (V, E)$ is a partition of V into disjoint *clusters*, such that the subgraph induced by each cluster is connected, the diameter of each cluster is $O(d(n))$, and the arboricity (and therefore the chromatic number) of the cluster graph is $O(c(n))$, where the cluster graph is obtained by contracting each cluster into a single node.

In order to apply the decomposition for design of efficient distributed algorithms, both the partitioning of the graph into clusters and a good coloring of the cluster graph must be efficiently computable. We show that any network has an $(n^{O(\sqrt{\frac{\log \log n}{\log n}})}, n^{O(\sqrt{\frac{\log \log n}{\log n}})})$ -decomposition and

¹A connected graph has arboricity k if its edges can be covered by at most k trees.

present a distributed algorithm which constructs such a decomposition in $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$ time. (Note that $n^{\sqrt{\frac{\log \log n}{\log n}}} = O(n^\epsilon)$ for any $\epsilon > 0$.)

We give several applications of our network decomposition technique.

- We give an $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$ -time algorithm for constructing a MIS in a distributed network.
- We show how to distributively color graphs with maximum degree Δ with $\Delta + 1$ colors in $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$ time.
- We present a deterministic algorithm for constructing a breadth-first search tree in an asynchronous distributed network in $O(m^{1+\frac{O(1)}{\sqrt[4]{\log n}}})$ messages and $O(D^{1+\frac{O(1)}{\sqrt[4]{\log n}}})$ time, where m is the number of edges in the network and D is the diameter of the network.

The MIS problem is a classical example of a problem for which obtaining efficient sequential algorithms is easy, but obtaining efficient parallel algorithms is much harder. Karp and Wigderson [11] have shown that the problem is in NC. A lot of research was done to improve the processor and time complexity of parallel algorithms for the MIS problem [10, 13, 14]. Unfortunately, direct distributed implementation of these algorithms is inefficient. Intuitively, the main problem is that, at each iteration, the algorithm decides on the next step by collecting the information from every node in the graph to a single memory location and making a choice based on this information, leading to maximum “benefit”. Hence, direct conversion of these algorithms to a distributed system leads to a running time that is at least linear in the diameter of the graph. (Observe that an MIS can be found by a trivial distributed algorithm in linear time.) For bounded degree, several distributed algorithms for MIS and related problems that run in $O(\log^* n)$ time were developed in [5, 7, 8].

The local nature of distributed networks is a limitation that often allows to prove lower bounds for them. Awerbuch [2] and Linial [12] show an $\Omega(\log^* n)$ lower bound on time needed to find an MIS in a distributed system. Note that the results mentioned in the previous paragraph imply that for constant degree graphs, these lower bounds are optimal to within a constant factor. For distributed networks of large degree, the gap between the upper and the lower bounds has been almost linear, namely $\frac{n}{\log^* n}$. The results of this paper decrease this gap to below n^ϵ for any $\epsilon > 0$.

The $\Delta + 1$ coloring problem is closely related to the MIS problem, and the algorithms for the former problem are closely related to ones for the latter problem and achieve similar bounds [7, 8, 9, 14].

The problem of constructing a breadth-first search tree in an asynchronous network (the *BFS problem*) is a fundamental problem in distributed computing, as it constitutes a bottleneck for many other algorithms, including *Spanning Tree*, *Leader Election*, computing a global function, etc. On a network with n nodes, m edges, and diameter D , the obvious lower bounds on the communication

and time complexity of the problem are $\Omega(m)$ and $\Omega(D)$, respectively. An existing *deterministic* BFS algorithm that is best in terms of *communication* is that of Awerbuch and Gallager [4]. This algorithm runs in $O(m^{1+\epsilon})$ messages and $O(n^{1+\epsilon})$ time, for any constant $\epsilon > 0$. An existing *deterministic* algorithm that is best in terms of *time* is obtained by combining a synchronizer [1] with a standard synchronous algorithm; the resulting algorithm runs in $O(m + nD \log_k n)$ messages and $O(kD)$ time, for any constant $k > O$. The first algorithm has a poor time complexity on networks of small diameter, while the second algorithm has a poor message complexity on sparse networks of large diameter. A *randomized* algorithm of Awerbuch [3] achieves a better message–time tradeoff. This algorithm runs in $O(m^{1+\epsilon})$ messages and $O(D^{1+\epsilon})$ time. We show how to apply techniques developed in this paper to make this algorithm deterministic, without degrading its asymptotic complexity.

2 Preliminaries

We consider the standard point-to-point communication network model (see *e.g.* [6] and [1]). The network topology is described by an undirected *communication graph* $G = (V, E)$, where the set of nodes represents processors of the network and the set of edges represents bidirectional communication channels between pairs of nodes. No common memory is shared by the processors. We denote the number of nodes in the network by n , and assume that each node has an ID represented by $O(\log n)$ bits.

We use the following complexity measures to evaluate the performance of distributed algorithms. The *communication complexity* is the worst-case total number of messages sent during an execution of the algorithm. The *time complexity* is the worst-case total number of time steps needed to complete the algorithm.

We describe the algorithms in terms of operations on *clusters* of nodes. We define *cluster* as follows.

Definition 1 A *cluster* is a connected component of the original graph G together with a spanning tree of the cluster and a node designated as the leader in the cluster. The *depth* of a cluster is the depth of the spanning tree. The ID of the cluster is the ID of its leader. Two clusters are neighbors in the *cluster graph* if there exists an edge in the original graph between a node in one cluster and a node in the other.

Given a set of clusters \mathcal{C} , we denote the graph induced by these clusters by $G[\mathcal{C}]$. We use $\delta_{\mathcal{C}}(X)$ to denote the degree of cluster X in the cluster graph $G[\mathcal{C}]$. We omit the subscript when the set \mathcal{C} is clear from the context.

To simplify the presentation, for the purpose of this extended abstract we assume that a message contains $O(n \log n)$ bits, and that each processor has $O(n^2 \log n)$ bits of memory. A more careful implementation of our algorithms uses $O(\log n)$ -bit messages and $O(d + n^{O(\frac{\log \log n}{\log n})})$ memory for

a processor whose degree in the network is d .

Suppose we are given a distributed algorithm and we want to convert it to run on a cluster graph, *i.e.* we want clusters to play the role of nodes from the point of view of this algorithm. We can achieve this by running the given algorithm on cluster leaders, and using the spanning trees of the clusters for the communication. Assuming sufficient message length and sufficient memory, it is easy to see that if T is the running time of the original algorithm and d is the upper bound on cluster depth, the simulation runs in $O(dT)$ time.

Except for the part dealing with the BFS problem, we assume that the communication network is synchronous, and that at each time step a processor can communicate with all of its neighbors. Awerbuch's Synchronizer- α [1] can be used to implement our algorithms on an asynchronous network. This does not change the asymptotic time bound T , but increases the asymptotic message complexity by an additive factor of mT . Observe that, since the algorithms considered in this paper require at least $\Omega(m)$ communication, the use of the synchronizer increases the communication complexity by at most an $O(m^\epsilon)$ factor, for any $\epsilon > 1$.

3 Basic Tools

In this section we describe several basic algorithms that are needed for our main result, the network decomposition algorithm, presented in Section 4.

Definition 2 Given a graph $G = (V, E)$ and a set $V' \subseteq V$, we say that a forest $\mathcal{F}_r = (V_r, E_r)$, $V' \subseteq V_r$, is (α, β) -ruling with respect to V' if the following three conditions hold:

1. The roots of the trees in \mathcal{F}_r are in V' .
2. The distance in G between roots of any two trees in \mathcal{F}_r is at least α .
3. The depth of each tree in the forest is at most β .

The next definition generalizes that of [5].

Definition 3 The set of tree roots of an (α, β) -ruling forest is called an (α, β) -ruling set.

We find a $(k, k \log n)$ -ruling forest by first finding a $(k, k \log n)$ -ruling set and then constructing a forest by executing a $(k \log n)$ -depth breadth-first search from each node in the ruling set. Figure 1 describes the RULING-SET algorithm that finds a $(k, k \log n)$ -ruling set with respect to a specified set of nodes $V' \subseteq V$ and an arbitrary k .

The algorithm RULING-SET starts by dividing the input set of vertices V' into two disjoint sets V_0 and V_1 according to the last bit of ID. Next we recursively find a $(k, k \log(n/2))$ -ruling set S_0 with respect to V_0 , and a $(k, k \log(n/2))$ -ruling set S_1 with respect to V_1 . Using breadth-first search that starts at each node in S_0 , we identify the nodes of S_1 that are at distance of less than k from nodes in S_0 , remove these nodes from S_1 , and return $S_0 \cup S_1$.

```

procedure RULING-SET( $G[V], V', k$ );

    Divide  $V$  into two disjoint sets  $V_0$  and  $V_1$  according to the last bits of IDs;
    Discard the last bits of IDs;
     $V'_0 \leftarrow V' \cap V_0$ ;
     $V'_1 \leftarrow V' \cap V_1$ ;
    for  $i \in \{0, 1\}$  in parallel do begin
         $S_i \leftarrow$  RULING-SET( $G[V_i], V'_i, k$ );
    end;
    for each  $v \in S_1$  in parallel do begin
        if there exists  $u \in S_1$  s.t. distance from  $v$  to  $u$  is at most  $k$ 
            then begin
                 $S_1 \leftarrow S_1 - v$ ;
            end;
    end;
     $S \leftarrow S_0 \cup S_1$ ;

    return ( $S$ );
end.

```

Figure 1: The RULING-SET algorithm.

Lemma 1 The algorithm RULING-SET on input $(G[V], V', k)$ produces a $(k, k \log n)$ -ruling set with respect to V' in $O(k \log n)$ time.

Proof: To prove correctness of the algorithm, note that by construction, the distance between any two nodes in S set is at least k . It remains to show that for any node $v \in V'$ that is not in the ruling set, there is a node w in the set, such that the distance between v and w is at most $k \log n$. We prove this by induction on the size of the graph. The statement is trivially true for graphs consisting of a single node. Assume that the algorithm works for graphs with at most 2^{i-1} nodes. By the induction hypothesis, given a graph with at most 2^i nodes, the recursive calls produce sets S_0 and S_1 such that each node $v \in V'$ is at distance at most $k(i-1)$ from a node w in one of these sets. Therefore, if w is in the returned set S , the induction hypothesis holds for v . Observe that w is not in S only if w is closer than k to some node $u \in S_0$. In this case, the distance from v to u is at most ik . Hence, any node in V' is at distance of at most $k \log n$ from some node in the constructed set S .

There are at most $O(\log n)$ bits in an ID and thus at most $O(\log n)$ levels of recursion, each level of recursion taking $O(k)$ time. ■

Next we describe the SPARSE-COLOR algorithm which colors a graph with maximum degree Δ with $(\Delta + 1)$ -colors. The algorithm is an adaptation of the algorithm of Goldberg, Plotkin, and Shannon [8].

The SPARSE-COLOR algorithm starts by dividing the nodes in V' into two disjoint sets V_0 and V_1 according to the last bit of the IDs. Then it recursively colors the graphs $G[V_0], G[V_1]$, induced

by these sets. The colors assigned to the nodes in V_0 remain, but the colors assigned to nodes in V_1 are used to determine a recoloring order on these nodes, such that all nodes recolored at time j are independent in G .

The following lemma follows directly from the algorithm description. (Note that we only need to know an upper bound on Δ in order to run this algorithm.)

Lemma 2 The algorithm `SPARSE-COLOR` runs in $O(\Delta \log n)$ time and produces a $(\Delta + 1)$ coloring of the input graph.

Another algorithm that we need in subsequent sections is the `MERGE-CLUSTERS` algorithm. The input to this algorithm is a forest in a cluster graph, and the output is a new cluster graph for the network, where the new clusters are formed by merging together the clusters in each one of the trees in the input forest. We omit the details of this algorithm. Note that if the maximum depth of the trees in the input forest is d and the maximum depth of any cluster in the input cluster graph was d' , then the maximum depth of any new cluster is bounded by $(2d + 1)d'$.

4 Network Decomposition

In this section we describe a distributed algorithm to decompose a network into node-disjoint clusters of small depth, such that the cluster graph can be colored with a small number of colors.

The `NETWORK-PARTITION` algorithm is described in Figure 2. The algorithm consists of two stages. The first stage divides the graph into clusters of small depth. The clusters are grouped into a small number of *levels*, where the degree of a cluster on level i in the cluster graph induced by the clusters on this and higher levels is smaller or equal to a predetermined parameter Δ^* . (We defer the discussion of the appropriate choice for the value of Δ^* until the end of the section.) The second stage colors the cluster graph produced by the first stage in $\Delta^* + 1$ colors using the fact that the cluster graph produced by the first stage has arboricity bounded by Δ^* .

Lemma 3 Consider a cluster X which belongs to \mathcal{C}_i for some $1 \leq i \leq L$. The degree of X in the cluster graph $G_i = G[\mathcal{C}_i, \mathcal{C}_{i+1}, \dots, \mathcal{C}_L]$ is bounded by Δ^* .

Proof: A cluster X belongs to \mathcal{C}_i only if it did not participate in merges at iteration i . In other words, the degree of X in the cluster graph \mathcal{C} considered at iteration i is bounded by Δ^* . Observe that since we never refine the decomposition, the degree of X in $G_i = G[\mathcal{C}_i, \mathcal{C}_{i+1}, \dots, \mathcal{C}_L]$ is at most its degree in the cluster graph \mathcal{C} . The lemma follows from the fact that the nodes that compose the cluster graph \mathcal{C} at iteration i are exactly the nodes that compose the cluster graph G_i at the end of the first stage. ■

Lemma 4 At the end of the first stage, each node belongs to some cluster $X \in \mathcal{C}_i$ for $1 \leq i \leq L$, for $L = \log n / \log \Delta^*$.

```

procedure NETWORK-PARTITION;

{Stage 1: Construct cluster graph}

  Make each node into a (trivial) cluster;
   $\mathcal{C} \leftarrow$  the set of (trivial) clusters;
   $L \leftarrow \log n / \log \Delta^*$ ;
  for  $i = 1$  to  $L$  do begin
     $\mathcal{C}' \leftarrow$  the set of clusters with degree at least  $\Delta^*$  in the cluster graph  $G[\mathcal{C}]$ ;
     $\mathcal{F}_r \leftarrow$  RULING-FOREST( $G[\mathcal{C}], \mathcal{C}', 3$ );
     $\mathcal{C}'' \leftarrow$  the set of clusters constructed by applying MERGE-CLUSTERS to the forest  $\mathcal{F}_r$ ;
     $\mathcal{C}_i \leftarrow$  clusters in  $\mathcal{C}$  not participating in the merge;
     $\mathcal{C} \leftarrow \mathcal{C}''$ ;
     $i \leftarrow i + 1$ ;
  end;

{Stage 2: Color cluster graph}

  for all  $1 \leq i \leq L$  in parallel do begin
     $\chi_i \leftarrow$  color  $G[\mathcal{C}_i]$  with  $\Delta^* + 1$  colors using SPARSE-COLOR;
  end;
  for  $i = L$  down to 1 do begin
    for  $j = 1$  to  $\Delta^* + 1$  do begin
      for all clusters  $X \in \mathcal{C}_i$  in parallel do begin
        if  $\chi_i(X) = j$ 
          then begin
             $\mathcal{C}_i^* \leftarrow \mathcal{C}_i \cup \mathcal{C}_{i+1} \cup \dots \cup \mathcal{C}_L$ ;
            recolor cluster  $X$  into color different from the color of its neighbors in  $G[\mathcal{C}_i^*]$ ;
          end;
        end;
      end;
    end;
  end;
end.

```

Figure 2: The NETWORK-PARTITION algorithm. The algorithm constructs the cluster graph $\overline{\mathcal{C}} = G[\mathcal{C}_1, \dots, \mathcal{C}_L]$ and colors the graph in $\Delta^* + 1$ colors.

Proof: Consider the cluster graph \mathcal{C} at iteration i . By construction, the distance in $G[\mathcal{C}]$ between any two roots of the trees in the forest computed during this iteration is three, and degree of each root of this forest in $G[\mathcal{C}]$ is at least Δ^* . Thus each one of the root clusters is merged with at least Δ^* other clusters. Since each non-root cluster is merged with some root cluster, the number of clusters decreases by a factor of at least Δ^* at each iteration, and the lemma follows. ■

Lemma 5 The maximum depth of a cluster produced by the NETWORK-PARTITION algorithm is bounded by $(9 \log n)^{(\log n / \log \Delta^*)}$.

Proof: Omitted. ■

Lemma 6 The NETWORK-PARTITION algorithm terminates in $O\left(\Delta^*(9 \log n)^{(\log n / \log \Delta^* + 1)}\right)$ time.

Proof: The running time of the first stage is determined by the last iteration of this stage, when clusters have the highest depth. By Lemma 5, the maximum depth of a cluster during this iteration is bounded by $O((9 \log n)^{(\log n / \log \Delta^*)})$. Together with Lemma 1, this implies $O((9 \log n)^{(\log n / \log \Delta^* + 1)})$ bound on the running time of the first stage.

The second stage starts with executing the SPARSE-COLOR algorithm. When this algorithm is applied, depth of the clusters is bounded by $O((9 \log n)^{(\log n / \log \Delta^*)})$. Lemma 2 implies the running time bound of $O(\Delta^*(9 \log n)^{(\log n / \log \Delta^* + 1)})$.

After executing the SPARSE-COLOR algorithm, the second stage proceeds from level to level, recoloring the clusters in each level. The total number of levels is $L = \log n / \log \Delta^*$, and for each level we execute Δ^* recoloring iterations, each iteration taking time proportional to the depth of the clusters at this level. Like in the analysis of the first stage, the running time of recoloring is determined by the time it takes to recolor the highest level. Hence, the recoloring takes $O(\Delta^*(9 \log n)^{(\log n / \log \Delta^*)})$ time. ■

By taking $\Delta^* = n^{\sqrt{\frac{\log \log n}{\log n}}}$, we get the following result. Notice that $n^{O(\sqrt{\frac{\log \log n}{\log n}})} = O(n^\epsilon)$ for any $\epsilon > 0$.

Theorem 1 In $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$ time it is possible to decompose the network into clusters of depth at most $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$, and to color the cluster graph with $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$ colors.

5 Applications

5.1 Maximal Independent Set

An MIS of nodes in a distributed network can be computed by the following algorithm. First, the algorithm applies NETWORK-PARTITION to construct a cluster graph \overline{C} and its coloring χ . Then, it iterates over the cluster colors. At each iteration, the algorithm finds an independent set of nodes in the original graph and deletes these nodes and their neighbors from the graph. At iteration i , the algorithm considers clusters colored χ_i , which are independent in the cluster graph. For each such cluster, the algorithm finds an MIS of a subgraph induced by undeleted nodes of the cluster. The union of these sets is an independent set such that, when nodes in this set and their neighbors are deleted from the graph, all nodes that are in clusters colored χ_i are deleted from the graph.

Theorem 2 A MIS in a distributed network can be found in $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$ time.

Proof: (sketch) Since the clusters produced by the network decomposition algorithm have small depth, we can adapt Luby’s MIS algorithm [14] so that each iteration runs in $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$ time. Luby’s MIS algorithm terminates in $O(\log^3 n)$ iterations. Hence, by Theorem 1, the algorithm runs in $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$ time. ■

5.2 $\Delta + 1$ Coloring

The linear-processor PRAM algorithm for $\Delta + 1$ coloring, due to Luby [14], can be changed to work when we start from a (legal) partial coloring of the input graph. This observation enables us to use the same strategy that we have used for the MIS problem. First, we use NETWORK-DECOMPOSITION algorithm to decompose the network into clusters, where the clusters are colored with $\Delta^* + 1$ colors. Then we proceed in iterations, where at iteration i we consider the nodes that comprise the clusters of the i th color, and use Luby’s coloring algorithm to color these.

Theorem 3 A maximum degree Δ graph can be distributively colored with $\Delta + 1$ colors in $n^{O(\sqrt{\frac{\log \log n}{\log n}})}$ time.

6 Deterministic BFS

6.1 Overview

The problem of constructing a $(3, 2)$ -ruling set has been the only obstacle towards eliminating randomness from (the best known) distributed algorithm for Breadth-First Search (BFS) and Shortest Paths [3]. The recursive version of that algorithm partitions the network into “strips” of $d \ll D$ successive BFS layers, and processes each strip one-by-one.

During the processing of a strip, we need to construct a $(3, 2)$ -ruling set of a certain “cluster graph”. Straightforward substitution of our MIS algorithm leads to a deterministic BFS algorithm that runs in $D^{1 + \frac{O(\sqrt[4]{\log \log n})}{\sqrt[4]{\log n}}}$ time, where D is the diameter of the graph. The disadvantage of this approach is that computing the MIS becomes the bottleneck for the BFS, making deterministic algorithm slower than the randomized one, which runs in $D^{1 + \frac{O(1)}{\sqrt[4]{\log n}}}$ time.

Reducing the running time of the deterministic algorithm to that of the randomized one requires a slight modification of the algorithm, combined with a more careful analysis. The modification is based on a new algorithm that constructs a $(3, 2)$ -ruling forest in a subgraph, given a spanning tree T that is not necessarily restricted to the sub-graph. The algorithm requires $O(\text{depth}(T) \cdot \log^2 n)$ time and $O((\tilde{m} + |T|) \log^2 n)$ messages. Here, $|T|$ is the cardinality of T , $\text{depth}(T)$ is its depth, and \tilde{m} is the number of edges in the sub-graph. There is no damage from dependance on the parameters

of the spanning tree T since during processing of the strip the algorithm anyway needs to perform (around) $n^{\frac{1}{\sqrt{\log_5 V}}}$ synchronizations thru the whole BFS tree.

In constructing this algorithm, we use the cost-benefit framework of Luby [14]. The time complexity and the communication complexity of the resulting BFS algorithm are same as in [3], namely $D^{1+\frac{O(1)}{\sqrt{\log n}}}$ and $m^{1+\frac{O(1)}{\sqrt{\log n}}}$, respectively. Below, we provide the details of the construction.

6.2 Radomized Algorithm

The randomized algorithm is given in Figure 3. In general, each vertex is classified to be one of four possible colors, *dark_green*, *light_green*, *yellow* and *red*. There is an implicit precedence on colors, *dark_green* has the highest precedence, followed by *light_green*, then *yellow* and finally *red*. Whenever we state that a vertex changes color, this is only true if the new color has higher precedence, e.g. a *dark_green* vertex always remains *dark_green*, whereas a *yellow* vertex can change color to *dark_green* or *light_green* but not to *red*. Initially, for all $v \in V$, $color(v) = red$. Upon termination, no *marked* vertex is colored *red*, i.e. all remaining *red* vertices are unmarked. The dominating set S is the set of all vertices with color *dark_green*. The colors have the following meanings:

dark_green: vertices in the dominating set.

light_green: neighbors of vertices in the dominating set (labeled *dark_green*).

yellow: neighbors of vertices labeled *light_green*.

red: all other vertices in the graph, i.e. vertices at distance bigger than two from vertices in the dominating set. The neighbors of all *red* vertices are either *red* or *yellow*.

```

For all  $v \in V$ ,  $color(v) \leftarrow red$ 
Do while red vertices exist in the marked set:                               /* This is the iteration loop */
  For all yellow and red vertices  $v$ ,
    Flip a coin with probability of heads  $\frac{1}{3d}$ 
    If  $coin = heads$  then  $color(v) \leftarrow dark\_green$ 
  For all yellow and red vertices  $v$ ,
    If there is a dark_green vertex adjacent to  $v$  then
       $color(v) \leftarrow light\_green$ 
  For all red vertices  $v$ ,
    If there is a light_green vertex adjacent to  $v$  then
       $color(v) \leftarrow yellow$ 
end
return the set of vertices colored dark_green

```

Figure 3: The randomized algorithm

6.3 Analysis of the randomized algorithm

Theorem 4 The randomized algorithm in Figure 3 terminates within $c \log |V|$ steps with probability at least $1 - 2^{-\Theta(c)}$. Furthermore, the expected number of *dark_green* vertices at termination is $O\left(\frac{|V| \log |V|}{d}\right)$.

Proof: Each marked *red* vertex is going to change color to *light_green* with probability at least some constant. The intuitive reasoning for this is that each marked *red* vertex has at least d neighbors, all of them colored *yellow* or *red*, and thus with some constant probability (since each such vertex is changing color to *dark_green* with probability $\frac{1}{3d}$) at least one of these neighbors changes color to *dark_green*. Thus, since a constant fraction of the marked *red* vertices change color at each iteration, the number of iterations before all marked *red* vertices are gone is $O(\log |V|)$. In each iteration, the expected number of vertices that are colored *dark_green* is $O\left(\frac{|V|}{d}\right)$, and thus overall the expected number of *dark_green* vertices is $O\left(\frac{|V| \log |V|}{d}\right)$. ■

7 The Deterministic Framework

Definition 4 Let *Red*, *Red_B* and *Yellow* denote the sets of *red*, *marked red*, and *yellow* vertices at the beginning of an iteration, respectively. For any set S , we denote by $\#S$ the cardinality of that set.

Our goal is to have a deterministic simulation of the algorithm described in Figure 3 such that at each iteration the following two properties hold:

1. The number of new *dark_green* vertices introduced by an iteration is at most $\frac{3(\#Yellow + \#Red)}{d}$.
2. The number of *marked red* vertices remaining upon termination of an iteration is at most $\frac{8(\#Red_B)}{9}$.

The general strategy for achieving this goal is to show that the probabilistic analysis holds when the random choices for the vertices are only pairwise independent, and in the analysis we use Profits and Costs as defined below. The strategy from there is to use the ideas in [14] to simulate the algorithm deterministically.

In the following analysis, we restrict attention to the subgraph induced on the union of the *yellow* and *red* vertices. Let $adj_v = \{u \mid (v, u) \in E\}$. For each *red* vertex v , let $subadj_v \subset adj_v$ such that $|subadj_v| = \tilde{d}$. Since each neighbor of a *red* v is either *red* or *yellow*, every vertex in $u \in subadj_v$ changes to *dark_green* with probability $\frac{1}{3d}$.

Let \hat{l} be a 0/1 labelling of the vertices in the graph, i.e. l_u is the label of vertex u . (The vertices labelled 1 become *dark_green*.) Let v be any *red* vertex. Define $I_v[\hat{l}] = 1$ if there is some $u \in \text{adj}_v$ such that $l_u = 1$ and $I_v[\hat{l}] = 0$ otherwise. ($I_v[\hat{l}] = 1$ implies that v changes color to *light_green*.) Define

$$Rbenefit_v[\hat{l}] = \sum_{u \in \text{subadj}_v} l_u \left(1 - \sum_{w \in \text{subadj}_v - \{u\}} l_w \right). \quad (1)$$

The importance of $Rbenefit_v[\hat{l}]$ is that it is a sum that only depends on products of at most two vertex labels, and that for all \hat{l} , $Rbenefit_v[\hat{l}] \leq I_v[\hat{l}]$. $E[Rbenefit_v[\hat{l}]]$ is an easily computable lower bound on the probability that v changes color to *light_green*, even when the coins of the vertices are only pairwise independent. Furthermore, it turns out that $E[Rbenefit_v[\hat{l}]$ is relatively close to the true probability that v changes color to *light_green*. Define

$$Rbenefit[\hat{l}] = \frac{\sum_{v \in Red_B} Rbenefit_v[\hat{l}]}{\#Red_B} \quad (2)$$

Because $Rbenefit_v[\hat{l}] \leq I_v[\hat{l}] \leq 1$, $Rbenefit[\hat{l}]$ is a lower bound on the fraction of *red* vertices that become *light_green* in this iteration. The idea is that the average value of $Rbenefit$ is going to be some constant, which helps to guarantee that property 2 holds. The fact that $Rbenefit[\hat{l}] \leq 1$ together with the $Gcost$ function defined in the next few paragraphs is going to help guarantee that property 1 holds.

In the following definitions, v is any vertex that is either *red* or *yellow*. Define

$$Gcost[\hat{l}] = \left(\sum_{v \in Red \cup Yellow} l_v \right) \cdot Gscale, \quad (3)$$

where $Gscale = \frac{\tilde{d}}{3(\#Red + \#Yellow)}$.

$Gcost[\hat{l}]$ is the number of *dark_green* vertices, times the “scale” factor $Gscale$. $Gcost$ is the fraction of all *red* and *yellow* vertices that become *dark_green* times the factor $\frac{\tilde{d}}{3}$. The reason for the factor $\frac{\tilde{d}}{3}$ is that we want to prevent the number of *dark_green* vertices from exceeding a $\frac{3}{\tilde{d}}$ fraction of all *red* and *yellow* vertices, since in this case $Gcost$ would outweigh $Rbenefit$.

Finally, we define

$$benefit[\hat{l}] = Rbenefit[\hat{l}] - Gcost[\hat{l}].$$

7.1 Implications of the average benefit

Recall that for each *red* vertex v , $|subadj_v| = \tilde{d}$.

$$E[Rbenefit[\hat{l}]] \geq \frac{\sum_{v \in Red_B} (\frac{\tilde{d}}{3\tilde{d}} - \frac{\tilde{d}^2}{\tilde{d}^2})}{\#Red_B} \geq \frac{2}{9}. \quad (4)$$

$$E[Gcost[\hat{l}]] = \left(\sum_{v \in Red \cup Yellow} \frac{1}{3\tilde{d}} \right) \cdot Gscale = \frac{1}{9}. \quad (5)$$

$$E[benefit[\hat{l}]] = E[Rbenefit[\hat{l}]] - E[Gcost[\hat{l}]] \geq \frac{1}{9}, \quad (6)$$

Theorem 5 For any \hat{l} , such that $benefit[\hat{l}] \geq \frac{1}{9}$, \hat{l} satisfies properties 1 and 2.

Proof: For all \hat{l} , $Rbenefit[\hat{l}] \leq 1$. For our choice of \hat{l} , $benefit[\hat{l}] > 0$ and thus

$$Gcost[\hat{l}] = Rbenefit[\hat{l}] - benefit[\hat{l}] \leq Rbenefit[\hat{l}] \leq 1. \quad (7)$$

However, the number of *red* and *yellow* vertices that become *dark_green* is $\frac{Gcost[\hat{l}]}{Gscale}$, which is at most $\frac{3(\#Red + \#Yellow)}{\tilde{d}}$. This completes the proof of property 1.

We proceed with the proof for property 2. $Rbenefit[\hat{l}] \geq benefit[\hat{l}] \geq \frac{1}{9}$ implies that the number of *marked red* vertices that become *light_green* is at least $\frac{\#Red_B}{9}$, thus proving property 2.

■

References

- [1] B. Awerbuch. Complexity of network synchronization. *J. Assoc. Comput. Mach.*, 32:804–823, 1985.
- [2] B. Awerbuch. A tight lower bound on the time of distributed maximal independent set algorithms. Unpublished manuscript, February 1987.
- [3] B. Awerbuch. Randomized Distributed Shortest Paths Algorithms. In *Proc. 21th ACM Symp. on Theory of Computing*, page (to appear), 1989.
- [4] B. Awerbuch and R. G. Gallager. A New Distributed Algorithm to Find Breadth First Search Trees. *IEEE Trans. Info. Theory*, IT-33:315–322, 1987.

- [5] R. Cole and U. Vishkin. Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 206–219, 1986.
- [6] R. G. Gallager, P. A. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems*, 5:66–77, 1983.
- [7] A. V. Goldberg and S. A. Plotkin. Parallel $(\Delta + 1)$ coloring of constant-degree graphs. *Information Processing Let.*, 25:241–245, 1987.
- [8] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel Symmetry-Breaking in Sparse Graphs. *SIAM J. Desc. Math.*, 1:434–446, 1989.
- [9] M. Goldberg. Parallel Algorithms for Three Graph Problems. Technical Report 86-4, RPI, 1986.
- [10] M. Goldberg and T. Spencer. A New Parallel Algorithm for the Maximal Independent Set Problem. In *Proc. 28th IEEE Symp. on Foundations of Comp. Sci.*, pages 161–165, 1987.
- [11] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 266–272, 1984.
- [12] N. Linial. Distributive Algorithms — Global Solutions from Local Data. In *Proc. 28th IEEE Symp. on Found. of Comp. Sci.*, pages 331–335, 1987.
- [13] M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM J. Comput.*, 15:1036–1052, 1986.
- [14] M. Luby. Removing Randomness in Parallel Computation without a Processor Penalty. In *Proc. 29th IEEE Symp. on Found. of Comp. Sci.*, pages 162–173, 1988.