Fast Local Computation Algorithms

Ronitt Rubinfeld[1] [*] Gil Tamir[2] [†] Shai Vardi[2] [‡] Ning Xie[1] [§]
[1]CSAIL, MIT, Cambridge MA 02139, USA
[2]School of Computer Science, Tel Aviv University, Israel
ronitt@csail.mit.edu giltamir@gmail.com shaivardi@gmail.com ningxie@csail.mit.edu

Abstract:
For input x, let $F(x)$ denote the set of outputs that are the "legal" answers for a computational problem F. Suppose x and members of $F(x)$ are so large that there is not time to read them in their entirety. We propose a model of *local computation algorithms* which for a given input x, support queries by a user to values of specified locations y_i in a legal output $y \in F(x)$. When more than one legal output y exists for a given x, the local computation algorithm should output in a way that is consistent with at least one such y. Local computation algorithms are intended to distill the common features of several concepts that have appeared in various algorithmic subfields, including local distributed computation, local algorithms, locally decodable codes, and local reconstruction.
We develop a technique, based on Beck's analysis in his algorithmic approach to the Lovász Local Lemma, which under certain conditions can be applied to construct polylogarithmic time local computation algorithms. We apply this technique to maximal independent set computations, scheduling radio network broadcasts, hypergraph coloring and satisfying k-SAT formulas.

Keywords: local computation algorithms; sublinear time algorithms; and the algorithmic Lovász Local Lemma

1 Introduction

Classical models of algorithmic analysis assume that the algorithm reads an input, performs a computation and writes the answer to an output tape. On massive data sets, such computations may not be feasible, as both the input and output may be too large for a single processor to process. Approaches to such situations range from proposing alternative models of computation, such as parallel and distributed computation, to requiring that the computations achieve only approximate answers or other weaker guarantees, as in sublinear time and space algorithms.

In this work, we consider the scenario in which only specific parts of the output $y = (y_1, \ldots, y_n)$ are needed at any point in time. For example, if y is a description of a maximal independent set (MIS) of a graph such that $y_i = 1$ if i is in the MIS and $y_i = 0$ otherwise, then it may be sufficient to determine $y_{i_1}, \ldots, y_{i_k}$ for a small number of vertices. To this end, we define a *local computation algorithm*, which supports queries such that after each query by the user to a specified location i, the local computation algorithm quickly outputs y_i. For problem statements that allow for more than one possible output for an input x, as is often the case in combinatorial search problems, the local computation algorithm must answer in such a way that is consistent with any past or future answers (in particular, there must always be at least one allowable output that is consistent with the answers of the local computation algorithm). For a given problem, the hope is that the complexity of a local computation algorithm is proportional to the amount of the solution that is requested by the user. Local computation algorithms are especially adapted to computations of good solutions of combinatorial search problems (cf. [26, 31, 32]).

Local computation algorithms are a formalization that describes concepts that are ubiquitous in the literature, and arise in various algorithmic subfields, including local distributed computation, local algorithms, locally decodable codes and local reconstruction models. We subsequently describe the relation between these models in more detail, but for now, suffice it to say that the models have some differences in the computations that they apply to (for example, whether they apply to function computations, search problems or approximation problems), the access to data that is allowed (for example, distributed computation models and other local algorithms models

[*] Supported by NSF grants 0732334 and 0728645, Marie Curie Reintegration grant PIRG03-GA-2008-231077 and the Israel Science Foundation grant nos. 1147/09 and 1675/09

[†] Supported by Israel Science Foundation grant no. 1147/09

[‡] Supported by Israel Science Foundation grant no. 1147/09

[§] Part of the work was done while visiting Tel Aviv University. Research supported in part by NSF Award CCR-0728645.

often specify that the input is a graph in adjacency list format), and the required running time bounds (whether the computation time is required to be independent of the problem size, or whether it is allowed to have computation time that depends on the problem size but has sublinear complexity). The model of local computation algorithms described here attempts to distill the essential common features of the aforementioned concepts into a unified paradigm.

After formalizing our model, we develop a technique that is specifically suitable for constructing polylogarithmic time local computation algorithms. This technique is based on Beck's analysis in his algorithmic approach to the Lovász Local Lemma (LLL) [8], and uses the underlying locality of the problem to construct a solution. All of our constructions must process the data somewhat so that Beck's analysis applies, and we use two main methods of performing this processing.

For maximal independent set computations and scheduling radio network broadcasts, we use a reduction of Parnas and Ron [32], which shows that, given a distributed network in which the underlying graph has degree bounded by a constant D, and a distributed algorithm (which may be randomized and only guaranteed to output an approximate answer) using at most t rounds, the output of any specified node of the distributed network can be simulated by a single processor with query access to the input with $O(D^{t+1})$ queries. For our first phase, we apply this reduction to a constant number of rounds of Luby's maximal independent set distributed algorithm [25] in order to find a partial solution. After a large independent set is found, in the second phase, we use the techniques of Beck [8] to show that the remainder of the solution can be determined via a brute force algorithm on very small subproblems.

For hypergraph coloring and k-SAT, we show that Alon's parallel algorithm for finding solutions guaranteed by the Lovász Local Lemma [2], can be used to give access to polylogarithmic sequential time queries regarding the coloring of a given node, or the setting of a given variable. For most of the queries, this is done by simulating the work of a processor and its neighbors within a small constant radius. For the remainder of the queries, we use Beck's analysis to show that the queries can be solved via the brute force algorithm on very small subproblems.

Note that parallel $O(\log n)$ time algorithms do not directly yield local algorithms under the reduction of Parnas and Ron [32]. Thus, we do not know how to apply our techniques to all problems with fast parallel algorithms, including certain problems in the work of Alon whose solutions are guaranteed by the Lovász Local Lemma and which have fast parallel algorithms [2]. Recently Moser and Tardos [28, 29] gave, under some slight restrictions, parallel algorithms which find solutions of all problems for which the existence of a solution is guaranteed by the Lovász Local Lemma [2, 8]. It has not yet been resolved whether one can construct local computation algorithms based on these more powerful algorithms.

1.1 Related work

Our focus on local computation algorithms is inspired by many existing works, which explicitly or implicitly construct such procedures. These results occur in a number of varied settings, including distributed systems, coding theory, and sublinear time algorithms.

Local computation algorithms are a generalization of *local algorithms*, which for graph theoretic problems represented in adjacency list format, produce a solution by adaptively examining only a constant sized portion of the input graph near a specified vertex. Such algorithms have received much attention in the distributed computing literature under a somewhat different model, in which the number of rounds is bounded to constant time, but the computation is performed by all of the processors in the distributed network [27, 30]. Naor and Stockmeyer [30] and Mayer, Naor and Stockmeyer [27] investigate the question of what can be computed under these constraints, and show that there are nontrivial problems with such algorithms. Several more recent works investigate local algorithms for various problems, including coloring, maximal independent set, dominating set (some examples are in [19–24]). Although all of these algorithms are distributed algorithms, those that use constant rounds yield (sequential) local computation algorithms via the previously mentioned reduction of Parnas and Ron [32].[1]

There has been much recent interest among the sublinear time algorithms community in devising local algorithms for problems on constant degree graphs and other sparse optimization problems. The goals of these algorithms have been to approximate quantities such as the optimal vertex cover, maximal matching, maximum matching, dominating set, sparse set cover, sparse packing and cover problems [15, 23, 26, 31, 32, 36]. One feature of these algorithms is that they show how to construct an oracle which for each vertex returns whether it part of the solution whose size is being approximated – for example, whether it is in the vertex cover or maximal matching. Their results show that this oracle can be implemented in time independent of the size of the graph (depending only

[1]Note that the parallel algorithms of Luby [25], Alon [2] and Moser and Tardos [29] do not automatically yield local algorithms via this transformation since their parallel runtimes are $t = \Omega(\log n)$.

on the maximum degree and the approximation parameter). However, because their goal is only to compute an approximation of their quantity, they can afford to err on a small fraction of their local computations. Thus, their oracle implementations give local computation algorithms for finding relaxed solutions to the the optimization problems that they are designed for. For example, constructing a local computation algorithm using the oracle designed for estimating the size of a maximal independent set in [26] yields a large independent set, but not necessarily a maximal independent set. Constructing a local computation algorithm using the oracle designed for estimating the size of the vertex cover [26, 31, 32] yields a vertex cover whose size is guaranteed to be only slightly larger than what is given by the 2-approximate algorithm being emulated – namely, by a multiplicative factor of at most $2 + \delta$ (for any $\delta > 0$).

Recently, local algorithms have been demonstrated to be applicable for computations on the web graph. In [6, 7, 9, 16, 34], local algorithms are given which, for a given vertex v in the web graph, computes an approximation to v's personalized PageRank vector and computes the vertices that contribute significantly to v's PageRank. In these algorithms, evaluations are made only to the nearby neighborhood of v, so that the running time depends on the accuracy parameters input to the algorithm, but there is no running time dependence on the size of the webgraph. Local graph partitioning algorithms have been presented in [7, 35] which find subsets of vertices whose internal connections are significantly richer than their external connections. The running time of these algorithms depends on the size of the cluster that is output, which can be much smaller than the size of the entire graph.

Though most of the previous examples are for sparse graphs or other problems which have some sort of sparsity, local computation algorithms have also been provided for problems on dense graphs. The property testing algorithms of [13] use a small sample of the vertices (a type of a coreset) to define a good graph coloring or partition of a dense graph. This approach yields local computation algorithms for finding a large partition of the graph and a coloring of the vertices which has relatively few edge violations.

The applicability of local computation algorithms is not restricted to combinatorial problems. One algebraic paradigm that has local computation algorithms is that of *locally decodable codes* [18], described by the following scenario: Suppose m is a string with encoding $y = E(m)$. On input x, which is close in Hamming distance to y, the goal of locally decodable coding algorithms is to provide quick access to the requested bits of m. More generally, the *reconstruction* models described in [1, 10, 33] describe scenarios where a string that has a certain property, such as monotonicity, is assumed to be corrupted at a relatively small number of locations. Let P be the set of strings that have the property. The reconstruction algorithm gets as input a string x which is close (in L_1 norm), to some string y in P. For various types of properties P, the above works construct algorithms which give fast query access to locations in y.

1.2 Organization

The rest of the paper is organized as follows. In Section 2 we present our computation model. Some preliminaries and notations that we use throughout the paper appear in Section 3. We then give local computation algorithms for the maximal independent set problem and the radio network broadcast scheduling problem in Section 4 and Section 5, respectively. In Section 6 we show how to use the parallel algorithmic version of the Lovász Local Lemma to give local computation algorithms for finding the coloring of nodes in a hypergraph. Finally, in Section 7, we show how to find settings of variables according to a satisfying assignment of a k-CNF formula.

2 Local Computation Algorithms: the model

We present our model of *local computation algorithms* for sequential computations of search problems, although computations of arbitrary functions and optimization functions also fall within our framework.

2.1 Model Definition

Definition 2.1. *For input x, let* $F(x) = \{y \mid y \text{ is a valid solution for input } x\}$. *The* search problem *is to find any $y \in F(x)$.*

In this paper, the description of both x and y are assumed to be very large.

Definition 2.2. *For $x, F(x)$ defined as above, a* $(t(n), \delta(n))$-local computation algorithm $\mathcal{A}$ *implements query access to an arbitrary $y \in F(x)$: More specifically, $\mathcal{A}$ is a (randomized) algorithm which gets a sequence of queries $i_1, \ldots, i_s$, and after each query i_j must produce an output y_{i_j} satisfying that the outputs $y_{i_1}, \ldots, y_{i_s}$ are substrings of some $y \in F(x)$. The probability of success over all s queries must be at least $1 - \delta(n)$. $\mathcal{A}$ has access to local computation memory on which it can perform current computations as well as store information from previous computations. We assume that x, the local computation tape and any random bits used are presented in the*

RAM word model, i.e., $\mathcal{A}$ is given the ability to access a word of any of these in one step. The running time of $\mathcal{A}$ on any query is at most $t(n)$. We say that $\mathcal{A}$ is a strongly local computation algorithm *if $t(n) \leq \operatorname{polylog}(n)$.*

Definition 2.3. *Let SLC be the class of problems that have strongly local computation algorithms.*

Note that when $|F(x)| > 1$, the y according to which $\mathcal{A}$ outputs may depend on the previous queries to $\mathcal{A}$ as well as any random bits available to $\mathcal{A}$. Also, we implicitly assume that the size of the output y is upper-bounded by some polynomial in $|x|$. The definition of local-computation algorithms rules out the possibility of their performing their computation by first computing the entire output. Analogous definitions can be made for a bit model. In principle, the model applies to general computations, including function computations, search problems and optimization problems of any type of object, and in particular, the input is not required by the model to be in a specific input format.

The model presented here is intended be more general, and thus differs from other local computation models in the following ways. First, queries and processing time have the same cost. Second, the focus is on problems with slightly looser running time bound requirements – poly-logarithmic dependence on the length of the input is desirable, but sublinear time in the length of the input is often nontrivial and can be acceptable. Third, the model places no restriction on the ability of the algorithm to access the input, as is the case in the distributed setting where the algorithm may only query nodes in its neighborhood (although such restrictions may be implied by the representation of the input). As such, the model may be less appropriate for certain distributed algorithms applications.

2.2 Relationship with other distributed and parallel models

A question that immediately arises is to characterize the problems to which the local-computation algorithm model applies. In this subsection, we note the relationship between problems solvable with local computation algorithms and those solvable with fast parallel or distributed algorithms.

From the work of [32] it follows that problems computable by fast distributed algorithms also have local computation algorithms.

Fact 2.4 ([32])**.** *If F is computable in $t(n)$ rounds on a distributed network in which the processor interconnection graph has bounded degree d, then F has a $d^{t(n)}$-local computation algorithm.*

Parnas and Ron ([32]) show this fact by observing that for any vertex v, if we run a distributed algorithm $\mathcal{A}$ on the subgraph $G_{k,v}$ (the vertices of distance at most k from v), then it makes the same decision about vertex v as it would if we would run D for k rounds on the whole graph G. They then give a reduction from randomized distributed algorithms to sublinear algorithms based on this observation.

Similar relationships hold in other distributed and parallel models, in particular, for problems computable by low depth bounded fan-in circuits.

Fact 2.5. *If F is computable by a circuit family of depth $t(n)$ and fan-in bounded by $d(n)$, then F has a $d(n)^{t(n)}$-local computation algorithm.*

Corollary 2.6. $NC^0 \subseteq SLC$.

In this paper we show solutions to several problems NC^1 via local computation algorithms. However, this is not possible in general as:

Proposition 2.7. $NC^1 \nsubseteq SLC$.

Proof. Consider the problem *n-XOR*, the XOR of n inputs. This problem is in NC^1. However, no sublinear time algorithm can solve *n-XOR* because it is necessary to read all n inputs. □

In this paper, we give techniques which allow one to construct local computation algorithms based on algorithms for finding certain combinatorial structures whose existence is guaranteed by constructive proofs of the LLL in [2, 8]. It seems that our techniques do not extend to all such problems.
An example of such a problem is *Even cycles in a balanced digraph*: find an even cycle in a digraph whose maximum indegree is not much greater that the minimum outdegree. Alon [2] shows that, under certain restriction on the input parameters, the problem is in NC^1. The local analogue of this question is to determine whether a given edge (or vertex) is part of an even cycle in such a graph. It is not known how to solve this quickly.

2.3 Locality-preserving reductions

In order to understand better which problems can be solved locally, we define *locality-preserving reductions*, which capture the idea that if problem B is locally computable, and problem A has a locality-preserving reduction to B then A is also locally computable.

Definition 2.8. *We say that A is $t(n)$*-locality-preserving reducible *to B via reduction F if F satisfies:*

1. $x \in A \iff F(x) \in B$.

2. *F is $t(n)$-locally computable, that is, every word of $F(x)$ can be computed by querying at most $t(n)$ words of x.*

Theorem 2.9. *If A is $t'(n)$-locality-preserving reducible to B and B is $t(n)$-locally computable, then A is $t'(n) \cdot t(n)$-locally computable.*

Proof. As A is $t'(n)$-locality-preserving reducible to B, to determine whether $x \in A$, it suffices to determine if $F(x) \in B$. Each word of $F(x)$ can be computed in time $t'(n)$, and we need to access at most $t(n)$ such words to determine whether $F(x) \in B$. □

3 Preliminaries

All logarithms in this paper are to the base 2. Let $\mathbb{N} = \{0, 1, \ldots\}$ denote the set of natural numbers. Let $n \geq 1$ be a natural number. We use $[n]$ to denote the set $\{1, \ldots, n\}$.

Unless stated otherwise, all graphs are undirected. Let $G = (V, E)$ be a graph. The *distance* between two vertices u and v in $V(G)$, denoted by $d_G(u, v)$, is the length of a shortest path between the two vertices. We write $N_G(v) = \{u \in V(G) : (u, v) \in E(G)\}$ to denote the neighboring vertices of v. Furthermore, let $N_G^+(v) = N(v) \cup \{v\}$. Let $d_G(v)$ denote the degree of a vertex v. Whenever there is no risk of confusion, we omit the subscript G from $d_G(u, v)$, $d_G(v)$ and $N_G(v)$.

The celebrated Lovász Local Lemma plays an important role in our results. We will use the simple symmetric version of the lemma.

Lemma 3.1 (Lovász Local Lemma [12]). *Let $A_1, A_2, \ldots, A_n$ be events in an arbitrary probability space. Suppose that the probability of each of these n events is at most p, and suppose that each event A_i is mutually independent of all but at most d of other events A_j. If $ep(d + 1) \leq 1$, then with positive probability none of the events A_i holds, i.e.,*

$$\Pr[\cap_{i=1}^{n} \bar{A}_i] > 0.$$

Several of our proofs use the following graph theoretic structure:

Definition 3.2 ([8]). *Define $W \subseteq V(G)$ to be a* 3-tree *if the pairwise distances of all vertices in W are each at least 3, and if the graph $G^* = (W, E^*)$ is connected, where E^* is the set of edges between each pair of vertices whose distance is exactly 3 in G.*

Let $\{E_n\}_{n \geq 0}$ be a sequence of events. We say that events $\{E_n\}$ happens *almost surely* if $\lim_{n \to +\infty} \Pr[E_n] = 1$; that is, asymptotically $\Pr[E_n] = 1 - o(1)$ as n tends to infinity.

4 Maximal Independent Set

An independent set (IS) of a graph G is a subset of vertices such that no two vertices are adjacent. An independent set is called a *maximal independent set* (MIS) if it is not properly contained in any other IS. It is well-known that a sequential greedy algorithm finds an MIS S in linear time: Order the vertices in G as $1, 2, \cdots, n$ and initialize S to the empty set; for $i = 1$ to n, if vertex i is not adjacent to any vertex in S, add i to S. The MIS obtained by this algorithm is call the *lexicographically first maximal independent set* (LFMIS). Cook [11] showed that deciding if vertex n is in the LFMIS is P-complete with respect to logspace reducibility. On the other hand, fast randomized parallel algorithms for MIS were discovered in 1980's [3, 17, 25]. The best known distributed algorithm for MIS runs in $O(\log^* n)$ rounds with a word-size of $O(\log n)$ [14]. By Fact 2.4, this implies a $d^{O(\log^* n \cdot \log n)}$ local computation algorithm. In this section, we give a local computation algorithm for MIS based on Luby's algorithm and techniques of Beck [8]. Our local computation algorithm runs in time $\mathrm{poly}(d) \cdot \log n + d^{O(d \log d)}$.

Let G be an undirected graph on n vertices and with maximum degree d. On input a vertex v, our algorithm decides whether v is in a maximal independent set using two phases. In Phase 1, we simulate Luby's parallel algorithm for MIS [25] via the reduction of [32]. That is, in each round, v tries to put itself into the IS with some small probability. It succeeds if none of its neighbors also tries to do the same. We run our Phase 1 algorithm for $O(d \log d)$ rounds. As it turns out, after Phase 1, most vertices have been either added to the IS or removed from the graph due to one (or more) of their neighbors being in the IS. Our key observation is that, following a variant of the argument of Beck [8], almost surely, all the connected components of the surviving vertices after Phase 1 have size at most $O(\log n)$. This enables us to perform the greedy algorithm for the connected component v lies in.

Our main result in this section is the following.

Theorem 4.1. *Let G be an undirected graph with n vertices and maximum degree d. Then there is a local computation algorithm which, on input a vertex v, decides if v is in a maximal independent set in time $d^{O(d \log d)} + \mathrm{poly}(d) \cdot \log n$. Moreover, the algorithm will give a consistent MIS for every vertex in G with probability at least $1 - 1/n$.*

4.1 Phase 1: simulating Luby's parallel algorithm

Figure 1 illustrates Phase 1 of our local computation algorithm for Maximal Independent Set. Our algorithm simulates Luby's algorithm for $r = O(d \log d)$ rounds. Every vertex v will be in one of three possible states:

- "selected" — v is in the MIS;
- "deleted" — one of v's neighbors is selected and v is deleted from the graph; and
- "$\perp$" — v is not in either of the previous states.

Initially, every vertex is in state "$\perp$". Once a vertex becomes "selected" or "deleted" in some round, it remains in that state in all the subsequent rounds.

The subroutine $\mathbf{MIS}(v, i)$ returns the state of a vertex v in round i. In each round, if vertex v is still in state "$\perp$", it "chooses" itself to be in the MIS with probability $1/2d$. At the same time, all its neighboring vertices also flip random coins to decide if they should "choose" themselves.[2] If v is the only vertex in $N^+(v)$ that is chosen in that round, we add v to the MIS ("select" v) and "delete" all the neighbors of v. However, the state of v in round i is determined not only by the random coin tosses of vertices in $N^+(v)$ but also by these vertices' states in round $i-1$. Therefore, to compute $\mathbf{MIS}(v, i)$, we need to recursively call $\mathbf{MIS}(u, i-1)$ for every $u \in N^+(v)$. [3] By induction, the total running time of simulating r rounds is $d^{O(r)} = d^{O(d \log d)}$.

If after Phase 1 no vertices remain (that is, all vertices are either "selected" or "deleted"), then the resulting independent set is a maximal independent set. Our main observation is, after simulating Luby's algorithm for $O(d \log d)$ rounds, we are already not far from a maximal independent set. Specifically, if vertex v returns "$\perp$" after Phase 1 of the algorithm, we call it a *surviving* vertex, and consider the subgraph induced on the surviving vertices. Following a variant of Beck's argument [8], we show that, almost surely, no connected component of surviving vertices is larger than $\mathrm{poly}(d) \cdot \log n$.

Let A_v be the event that vertex v is a surviving vertex. Note that event A_v depends on the random coin tosses v and v's neighborhood of radius r made during the first r rounds, where $r = O(d \log d)$. To get rid of the complication caused by this dependency, we consider another set of events generated by a related random process.

Consider a variant of our algorithm $\mathbf{MIS}$, which we call $\mathbf{MIS}_B$. The two algorithms are identical except that, in $\mathbf{MIS}_B$, when a vertex v is "selected", we mark v but do not remove v or its neighboring vertices from the graph.

Instead, we keep them there until the end of the algorithm, and, although they are "selected" or "deleted", they can still "choose" themselves and thus prevent their neighbors from being "selected". We let B_v be the event that v is never marked during all r rounds of $\mathbf{MIS}_B$.

Claim 4.2. *$A_v \subset B_v$ for every vertex v.*

Proof. This follows directly from the fact that a necessary condition for A_v to happen is v never get "selected" in any of the r rounds. □

As a simple corollary, we have

Corollary 4.3. *For any vertex set $W \subset V(G)$, $\Pr[\cap_{v \in W} A_v] \leq \Pr[\cap_{v \in W} B_v]$.*

A graph H on the vertices $V(G)$ is called a *dependency graph* for $\{B_v\}_{v \in V(G)}$ if for all v the event B_v is mutually independent of all B_u with $(u, v) \notin H$.

The following two claims are identical to Claim 5.6 and Claim 5.7 in Section 5 respectively, we therefore omit the proofs.

Claim 4.4. *The dependency graph H has maximum degree d^2.*

Claim 4.5. *For every $v \in V$, the probability that B_v occurs is at most $1/2d^6$.*

Now we are ready to prove the main lemma for our local computation algorithm for MIS.

Lemma 4.6. *After Phase 1, with probability at least $1 - 1/n^2$, all connected components of the surviving vertices are of size at most $O(\mathrm{poly}(d) \cdot \log n)$.*

Proof [Sketch]: The proof is very similar to that of Lemma 5.8, the only difference is, for a 3-tree W, we use the following inequalities

$$\begin{aligned} & \Pr[\text{all vertices in } W \text{ are surviving vertices}] \\ &= \Pr[\cap_{v \in W} A_v] \leq \Pr[\cap_{v \in W} B_v] \\ &= \prod_{v \in W} \Pr[B_v] \\ &\leq \left(\frac{1}{2d^6}\right)^{|W|} \end{aligned}$$

to bound the expected number of 3-trees of surviving vertices. We omit the rest of the proof. □

[2]We store all the randomness generated by each vertex in each round so that our answers will be consistent. However, we generate the random bits only when the state of corresponding vertex in that round is requested.

[3]A subtle point in the subroutine $\mathbf{MIS}(v, i)$ is that when we call $\mathbf{MIS}(v, i)$, we only check if vertex v is "selected" or not in round i. If v is "deleted" in round i, we will not detect this until we call $\mathbf{MIS}(v, i+1)$, which checks if some neighboring vertex of v is selected in round i. However, such "delayed" decisions will not affect our analysis of the algorithm.

```
MAXIMAL INDEPENDENT SET: PHASE 1
Input: a graph $G$ and a vertex $v \in V$
Output: {"true", "false", "$\perp$"}
    For $i$ from 1 to $r = 24d \log d$
        (a) If $\mathbf{MIS}(v, i) =$ "selected"
            return "true"
        (b) Else if $\mathbf{MIS}(v, i) =$ "deleted"
            return "false"
        (c) Else
            return "$\perp$"

$\mathbf{MIS}(v, i)$
Input: a vertex $v \in V$ and a round number $i$
Output: {"selected", "deleted", "$\perp$"}
1.  If $v$ is marked "selected" or "deleted"
        return "selected" or "deleted", respectively
2.  For every $u$ in $N(v)$
        If $\mathbf{MIS}(u, i-1) =$ "selected"
            mark $v$ as " deleted" and return "deleted"
3.  $v$ chooses itself independently with probability $\frac{1}{2d}$
        If $v$ chooses itself
            (i) For every $u$ in $N(v)$
                If $u$ is marked "$\perp$", $u$ chooses itself independently with probability $\frac{1}{2d}$
            (ii) If $v$ has a chosen neighbor
                return "$\perp$"
            (iii) Else
                mark $v$ as "selected" and return "selected"
        Else
            return "$\perp$"
```

Figure 1: Local Computation Algorithm for MIS: Phase 1

4.2 Phase 2: Greedy search in the connected component

If v is a surviving vertex after Phase 1, we perform Phase 2 of the algorithm. In this phase, we first explore v's connected component, $C(v)$, in the graph induced on G by all the vertices in state "$\perp$". If the size of $C(v)$ is larger than $c_2 \log n$ for some constant c_2, we abort and output "Fail". Otherwise, we perform the simple greedy algorithm described at the beginning of this section to find the MIS in $C(v)$. The running time for Phase 2 is at most $O(|C(v)|) \leq \mathrm{poly}(d) \cdot \log n$. Therefore, the total running time of our local computation algorithm for MIS is $d^{O(d \log d)} + \mathrm{poly}(d) \cdot \log n$.

5 Radio Networks

For the purposes of this section, a *radio network* is an undirected graph $G = (V, E)$ with one processor at each vertex. The processors communicate with each other by transmitting messages in a synchronous fashion to their neighbors. In each round, a processor P can either receive a message, send messages to all of its neighbors, or do nothing. We will focus on the radio network that is referred to as a *Type II network*:[4] in [2]. P receives a message from its neighbor Q if P is silent, and Q is the only neighbor of P that transmits in that round. Our goal is to check whether there is a two-way connection between each pair of adjacent vertices. To reach this goal, we would like to find a schedule such that each vertex in G broadcasts in one of the K rounds and K is as small as possible.

Definition 5.1 (Broadcast function)**.** *Let $G = (V, E)$ be an undirected graph. We say $F_r : V \to [K]$ is a* broadcast function *for the network G if the following holds:*

1. *Every vertex v broadcasts once and only once in round $F_r(v)$ to all its neighboring vertices;*
2. *No vertex receives broadcast messages from more than one neighbor in any round;*
3. *For every edge $(u, v) \in G$, u and v broadcast in distinct rounds.*

Let Δ be the maximum degree of G. Alon et. al. [4, 5] show that the minimum number of rounds K satisfies $K = \Theta(\Delta \log \Delta)$. Furthermore, Alon [2] gives an NC_1 algorithm that computes the broadcast function F_r with $K = O(\Delta \log \Delta)$. Here we give a local computation algorithm for this problem, i.e. given a degree-bounded graph $G = (V, E)$ in the adjacency list form and a vertex $v \in V$, we output the round number in which v broadcasts in logarithmic time. Our solution is *consistent* in that the answers the algorithm outputs to the various $v \in V$ agree with some broadcast scheduling function F_r.

Let $G^{1,2}$ be the "square graph" of G; that is, u and v are connected in $G^{1,2}$ if and only if their distance in G is either one or two. Our algorithm is based on finding an *independent set cover* of $G^{1,2}$ which simulates Luby's Maximal Independent Set algorithm [25]. Note that if we denote the maximum degree of $G^{1,2}$ by d, then $d \leq \Delta^2$.

Definition 5.2 (Independent Set Cover)**.** *Let $H = (V, E)$ be an undirected graph. A collection of vertex subsets $\{S_1, \dots, S_t\}$ is an* independent set cover *(ISC) for H if these vertex sets are pairwise disjoint, each S_i is an independent set in H and their union equals V. We call t the* size *of ISC $\{S_1, \dots, S_t\}$.*

Fact 5.3. *If $\{S_1, \dots, S_t\}$ is an ISC for $G^{1,2}$, then the function defined by $F_r(v) = i$ iff $v \in S_i$ is a broadcast function.*

Proof. First note that, since the union of $\{S_i\}$ equals V, $F_r(v)$ is well-defined for every $v \in G$. That is, every v broadcasts in some round in $[t]$, hence both directions of every edge are covered in some round. As v can only be in one IS, it only broadcasts once. Second, for any two vertices u and v, if $d(u, v) \geq 3$, then $N(u) \cap N(v) = \emptyset$. It follows that, if in each round all the vertices that broadcast are at least 3-apart from each other, no vertex will receive more than one message in any round. Clearly the vertices in an independent set of $G^{1,2}$ have the property that all the pairwise distances are at least 3. □

The following is a simple fact about ISCs.

Fact 5.4. *For every undirected graph H on n vertices with maximum degree d, there is an ISC of size at most d. Moreover, such an ISC can be found by a greedy algorithm in time at most $O(dn)$.*

Proof. We repeatedly apply the greedy algorithm that finds an MIS in order to find an ISC. Recall that the greedy algorithm repeats the following until the graph has no unmarked vertex: pick an unmarked vertex v, add it to the IS and mark off all the vertices in $N(v)$. Clearly each IS found by the greedy algorithm has size at least $\frac{n}{d+1}$. To partition the vertex set into an ISC, we run this greedy algorithm to find an IS which we call S_1, and delete all the vertices in S_1 from the graph. Then we run the greedy algorithm on the new graph again to get S_2, and so on. After running at most d rounds (since each round reduces the maximum degree of the graph by at least one), we partition all the vertices into an ISC of size at most d and the total running time is at most $O(dn)$. □

[4]The other model, Type I radio network, is more restrictive: A processor P receives a message from its neighbor Q in a given round only if P is silent, Q transmits and P chooses to receive from Q in that round.

Our main result in this section is a local computation algorithm that computes an ISC of size $O(d \log d)$ for any graph of maximum degree d. On input a vertex v, our algorithm outputs the index i of a vertex subset S_i to which v belongs, in an ISC of H. We will call i the *round number* of v in the ISC. By Fact 5.3, applying this algorithm to graph $G^{1,2}$ gives a local computation algorithm that computes a broadcast function for G.

5.1 A local computation algorithm for ISC

Our main result for computing an ISC is summarized in the following theorem.

Theorem 5.5. *Let H be an undirected graph on n vertices with maximum degree d. Then there is a local computation algorithm which, on input a vertex v, computes the round number of v in an ISC of size at most $O(d \log d)$ in time $\operatorname{poly}(d) \cdot \log n$. Moreover, the algorithm will give a consistent ISC for every vertex in H with probability at least $1 - 1/n$.*

On input a vertex v, our algorithm computes the round number of v in two phases. In Phase 1 we simulate Luby's algorithm for MIS [25] for $O(d \log d)$ rounds. At each round, v tries to put itself in the independent set generated in that round. That is, v chooses itself with probability $1/2d$ and if none of its neighbors choose themselves, then v is selected in that round and we output that round number for v. As we show shortly, after Phase 1, most vertices will be assigned a round number. We say v *survives* if it is not assigned a round number. We consider the connected component containing v after one deletes all vertices that do not survive from the graph. Following an argument similar to that of Beck [8], almost surely, all such connected components of surviving vertices after Phase 1 have size at most $O(\log n)$. This enables us, in Phase 2, to perform the greedy algorithm on v's connected component to deterministically compute the round number of v in time $O(\log n)$.

5.1.1 Phase 1 algorithm

Phase 1 of our local computation algorithm for computing an ISC is shown in Figure 2. [5]

For every $v \in V$, let A_v be the event that vertex v returns "$\perp$", i.e. v is not selected after r rounds. We call such a v a *surviving* vertex. After deleting all v that do not survive from the graph, we are interested in bounding the size of the largest remaining connected component. Clearly event A_v depends on the random coin tosses of v and v's neighboring vertices in all the r rounds. A graph H on the vertices $V(H)$ (the indices for the A_v) is called a *dependency graph* for $\{A_v\}_{v \in V(H)}$ if for all v the event A_v is mutually independent of all A_u with $(u, v) \notin H$.

Claim 5.6. *The dependency graph H has maximum degree d^2.*

Proof. Since for every vertex v, A_v depends only on the coin tosses of v and vertices in $N(v)$ in each of the r rounds, the event A_v is independent of all A_u such that $d_H(u, v) \geq 3$. The claim follows as there are at most d^2 vertices at distance 1 or 2 from v. □

Claim 5.7. *For every $v \in V$, the probability that A_v occurs is at most $1/2d^6$.*

Proof. The probability that vertex v is chosen in round i is $\frac{1}{2d}$. The probability that none of its neighbors is chosen in this round is $(1-\frac{1}{2d})^{d(v)} \geq (1-\frac{1}{2d})^d \geq 1/2$. Since the coin tosses of v and vertices in $N(v)$ are independent, the probability that v is selected in round i is at least $\frac{1}{2d} \cdot \frac{1}{2} = \frac{1}{4d}$. We get that the probability that A_v happens is at most $(1 - \frac{1}{4d})^{24d \log d} \leq \frac{1}{2d^6}$. □

The following observation is crucial in our local computation algorithm.

Lemma 5.8. *After Phase 1, with probability at least $1 - 1/n^2$, all connected components of the surviving vertices are of size at most $O(\operatorname{poly}(d) \cdot \log n)$.*

Proof. The proof is similar to that of Beck [8].

We bound the number of 3-trees in H (the dependency graph for events $\{A_v\}$) of 3-trees of size w as follows.

Let H^3 denote the "distance-3" graph of H, that is, vertices u and v are connected in H^3 if their distance in H is exactly 3. Since every 3-tree is a connected subgraph in H^3, the number of 3-trees is at most the number of connected subgraphs in H^3. Note that H^3 has maximum degree $D = d(d-1)^2$. It is easy to verify that the number of connected subgraphs of size w in a D-regular graph on n vertices is at most $O(nD^{2w})$. It follows that the number of 3-trees of size w in H^3 is at most $O(nd^{6w})$. Since all vertices in W are at least 3-apart, all the events $\{A_v\}_{v \in W}$ are mutually independent. That is,

$$\Pr[\text{all vertices in } W \text{ are surviving vertices}] = \prod_{v \in W} \Pr[A_v] \leq \left(\frac{1}{2d^6}\right)^w$$

[5]In 2(b), we flip random coins for u even if u is selected in a previous round. We do this for the technical reason that we want to rid the dependency of v on nodes that are not neighbors to simplify our analysis. Thus our analysis is overly pessimistic since if selected neighbors stop choosing themselves, it only increases the chance of v being selected.

```
INDEPENDENT SET COVER: PHASE 1
Input: a graph $H$ and a vertex $v \in V$
Output: the round number of $v$ in the ISC or “$\perp$”
1.  Initialize all vertices in $N^+(v)$ to state “$\perp$”
2.  For $i = 1$ to $r = 24d \log d$
        (a) If $v$ is labeled "$\perp$"
                $v$ chooses itself independently with probability $\frac{1}{2d}$
        (b) If $v$ chooses itself
                (i) For every $u \in N(v)$
                    (even if $u$ is labeled “selected in round $j$”
                    for some $j < i$, we still flip random coins for it)
                    $u$ chooses itself independently with probability $\frac{1}{2d}$
                (ii) If $v$ has a chosen neighbor,
                    $v$ unchooses itself
                (iii) Else
                    $v$ is labeled “selected in round $i$”
                    return $i$
3.  return “$\perp$”
```

Figure 2: Algorithm for finding an Independent Set Cover: Phase 1.

by Claim 5.7. Now the expected number of 3-trees of size w is at most

$$nd^{6w}\left(\frac{1}{2d^6}\right)^w = n2^{-w} \leq 1/n^2,$$

for $w = c_1 \log n$, where c_1 is some constant. By Markov's inequality, with probability at least $1 - 1/n^2$, there is no 3-tree of size larger than $c_1 \log n$. Finally, by a simple variant of the 4-tree Lemma in [8] (that is, instead of the “4-tree lemma”, we need a “3-tree lemma” here), a connected component of size s in H contains a 3-tree of size at least s/d^3 and the lemma follows. □

5.1.2 Phase 2 algorithm

If v is a surviving vertex after Phase 1, we perform Phase 2 of the algorithm. In this phase, we first explore the connected component, $C(v)$, that the surviving vertex v lies in. If the size of $C(v)$ is larger than $c_2 \log n$ for some constant c_2, we abort and output “Fail”. Otherwise, we perform the simple greedy algorithm described in Fact 5.4 to partition $C(v)$ into at most d subsets deterministically. The running time for Phase 2 is at most $\mathrm{poly}(d) \cdot \log n$. Since any independent set of a connected component can be combined with independent sets of other connected components to form an IS for the surviving vertices, we conclude that the total size of ISC we find is $O(d \log d) + d = O(d \log d)$.

5.2 Discussions

Now a simple application of Theorem 5.5 to $G^{1,2}$ gives a local computation algorithm for the broadcast function.

Theorem 5.9. *Given a graph $G = (V, E)$ with n vertices and maximum degree Δ and a vertex $v \in V$, there exists a randomized local computation algorithm running in time at most $\mathrm{poly}(d) \cdot \log n$ that computes a broadcast function with at most $O(\Delta^2 \log \Delta)$ rounds. Furthermore, with probability at least $(1 - 1/n^2)$, the broadcast function it outputs is consistent for all queries to the vertices of the graph.*

We note our round number bound is quadratically larger than that of Alon's parallel algorithm [2]. We do not know how to turn his algorithm into a local computation algorithm.

6 Hypergraph two-coloring

A *hypergraph* H is a pair $H = (V, E)$ where V is a finite set whose elements are called *nodes* or *vertices*, and E is a family of non-empty subsets of V, called *hyperedges*. A hypergraph is called *k-uniform* if each of its hyperedges contains precisely k vertices. A *two-coloring* of a hypergraph H is a mapping $\mathbf{c} : V \to \{\text{red, blue}\}$ such that no hyperedge in E is monochromatic. If such a coloring exists, then we say H is *two-colorable*. We assume that each hyperedge in H intersects at most d other hyperedges. Let N be the number of hyperedges in H. Here we think of k and d as fixed constants and all asymptotic forms are with respect to N. By the Lovász Local Lemma, when $e(d+1) \leq 2^{k-1}$, the hypergraph H is two-colorable.

Let m be the total number of vertices in H. Note that $m \leq kN$, so $m = O(N)$. For any vertex $x \in V$, we use $\mathcal{E}(x)$ to denote the set of hyperedges x belongs to. For convenience, for any hypergraph $H = (V, E)$, we define an m-by-N *vertex-hyperedge incidence matrix* $\mathcal{M}$ such that, for any vertex x and hyperedge e, $\mathcal{M}_{x,e} = 1$ if $e \in \mathcal{E}(x)$ and $\mathcal{M}_{x,e} = 0$ otherwise. A natural representation of the input hypergraph H is this vertex-hyperedge incidence matrix $\mathcal{M}$. Moreover, since we assume both k and d are constants, the incidence matrix $\mathcal{M}$ is necessarily very sparse. Therefore, we further assume that the matrix $\mathcal{M}$ is implemented via linked lists for each row (that is, vertex x) and each column (that is, hyperedge e).

Let G be the *dependency graph* of the hyperedges in H. That is, the vertices of the undirected graph G are the N hyperedges of H and a hyperedge E_i is connected to another hyperedge E_j in G if $E_i \cap E_j \neq \emptyset$. It is easy to see that if the input hypergraph is given in the above described representation, then we can find all the neighbors of any hyperedge E_i in the dependency graph G (there are at most d of them) in $O(\log N)$ time.

6.1 Our main result

A natural question to ask is: Given a two-colorable hypergraph H, and a vertex v, can we *quickly* compute the coloring of v? Our main result in this section is, given a two-colorable hypergraph H whose two-coloring scheme is guaranteed by the Lovász Local Lemma (with slightly weaker parameters), we give a local computation algorithm which answers queries of the coloring of any single vertex in $\mathrm{polylog}(N)$ time, where N is the number of the hyperedges in H. The coloring returned by our oracle will agree with some two-coloring of the hypergraph with probability at least $1 - o(1)$.

Theorem 6.1. *Let d and k be such that there exist three positive integers k_1, k_2 and k_3 such that the followings hold:*

$$\begin{aligned} k_1 + k_2 + k_3 &= k, \\ 16d(d-1)^3(d+1) &< 2^{k_1}, \\ 16d(d-1)^3(d+1) &< 2^{k_2}, \\ 2e(d+1) &< 2^{k_3}. \end{aligned}$$

Then there exists a local computation algorithm which, given a hypergraph H and any sequence of queries to the colors of vertices $(x_1, x_2, \ldots, x_s)$, with probability at least $1 - 1/N^2$, returns a consistent coloring for all x_i's which agrees with a 2-coloring of H. Moreover, the algorithm answers each single query in expected $O((\log N)^c)$ *time, where c is some constant (depending only on k and d).*

6.2 Overview of the coloring algorithm

Our local computation algorithm imitates the parallel coloring algorithm of Alon [2]. Recall that Alon's algorithm runs in three phases. In the first phase, we randomly color each vertex in the hypergraph. If some hyperedge has k_1 vertices in one color and no vertices in the other color, we call it a *dangerous* edge and mark all the remaining vertices in that hyperedge as *troubled*. After the first phase, we delete all hyperedges which have been assigned both colors and call the remaining hyperedges *surviving* edges. Now we repeat the same process again, but this time a hyperedge becomes dangerous if $k_1 + k_2$ vertices are colored the same and no vertices are colored by the other color. Finally, in the third phase, we do a brute-force search for a coloring in each of the connected components of the surviving vertices as they are of size $O(\log \log N)$ almost surely.

A useful observation is, in the first phase of Alon's algorithm, we can color the vertices in *arbitrary* order. In particular, this order can be taken to be the order that queries to the local computation algorithm are made in. If the coloring of a vertex x can not be determined in the first phase, then we explore the dependency graph around the hyperedges containing x and find the connected component of the *surviving* hyperedges to perform the second phase coloring. To ensure that we always find a *consistent* coloring, we repeat the second phase colorings many times until all the connected components resulting from the second phase coloring are of size at most $O(\log \log N)$. If that still can not decide the coloring of x, then we run the third (and final) phase of coloring, in which we exhaustively search for a two-coloring for vertices in some (small, i.e., of size at most $O(\log \log N)$) connected component in G as guaranteed by our second phase coloring. Following Alon's analysis, for any vertices in H, the total time running all these three phases is $\mathrm{polylog}(N)$ almost surely.

During the execution of the algorithm, each hyperedge will be in either *initial*, *safe*, *unsafe*-1, *unsafe*-2, *dangerous*-1 or *dangerous*-2 state. Vertices will be in either *uncolored*, *red*, *blue*, *trouble-vertex*-1 or *trouble-vertex*-2 state. The meanings of all these states should be clear from their names. Initially every hyperedge is in *initial* state and every vertex is in *uncolored* state.

```
**Phase-**1 **Coloring**($x$)
Input: a vertex $x \in V$
Output: a color in {*red*, *blue*}
1. If $x$ is already colored
       Return the color of $x$
2. If $x$ is in *trouble-vertex*-1 state
       Return **Phase-**2 **Coloring**($x$)
3. If $x$ is in *trouble-vertex*-2 state
       Return **Phase-**3 **Coloring**($x$)
4. If $x$ is in *uncolored* state
       (a) Uniformly at random choose a color $c$ for $x$ from {*red, blue*}
       (b) Update the states of all hyperedges in $\mathcal{E}(x)$
       (c) Return color $c$
```

Figure 3: Phase 1 coloring algorithm

6.3 Phase 1 coloring

If the color of vertex x is queried, we first check the current state of x. If x is already colored (that is, x is in either *red* or *blue* state), then we simply return that color. If x is in the *trouble-vertex*-1 state, we invoke Phase 2 coloring for vertex x. If x is in the *trouble-vertex*-2 state, we invoke Phase 3 coloring for vertex x. If x is *uncolored*, then we flip a fair coin to color x red or blue with equal probability (that is, vertex x's state becomes *red* or *blue*, respectively). After that, we update the status of all the hyperedges in $\mathcal{E}(x)$. Specifically, if some $E_i \in \mathcal{E}(x)$ has k_1 vertices in one color and no vertices in the other color, then we change E_i from *initial* into *dangerous*-1 state. Furthermore, all uncolored vertices in E_i will be changed to *trouble-vertex*-1 states. On the other hand, if both colors appear among the vertices of E_i, we update the state of E_i from *initial* to *safe*. If none of the vertices in a hyperedge is *uncolored* and the hyperedge is still in *initial* state (that is, it is neither *safe* or *dangerous*-1), then we change its state to *unsafe*-1. Note that if a hyperedge is *unsafe*-1 then all of its vertices are either colored or in *trouble-vertex*-1 state, and the colored vertices are monochromatic.

Running time analysis. The running time of Phase 1 coloring for an *uncolored* vertex x is $O(kd) = O(1)$ (recall that we assume both k and d are constants). This is because vertex x can belong to at most $d+1$ hyperedges, hence there are at most $k(d+1)$ vertices that need to be updated during Phase 1. If x is already a *colored* vertex, the running time is clearly $O(1)$. Finally, the running time of Phase 1 coloring for a *trouble-vertex*-1 or *trouble-vertex*-2 vertex is $O(1)$ plus the running time of Phase 2 coloring or $O(1)$ plus the running time of Phase 3 coloring, respectively.

6.4 Phase 2 coloring

During the second phase of coloring, given an input vertex x (which is necessarily a *trouble-vertex*-1), we first explore the dependency graph G of the hypergraph H by keep coloring some other vertices whose colors may have some correlation with the coloring of x. In doing so, we grow a connected component of *surviving*-1 hyperedges containing x in G. Here, a hyperedge is called *surviving*-1 if it is either *dangerous*-1 or *unsafe*-1. We denote this connected component of *surviving*-1 hyperedges surrounding vertex x by $C_1(x)$.

Growing the connected component. Specifically, in order to find out $C_1(x)$, we maintain a set of hyperedges $\mathcal{E}_1$ and a set of vertices V_1. Throughout the process of exploring G, V_1 is the set of *uncolored* vertices that are contained in some hyperedge in $\mathcal{E}_1$. Initially $\mathcal{E}_1 = \mathcal{E}(x)$. Then we independently color each vertex in V_1 *red* or *blue* uniformly at random. After coloring each vertex, we update the state of every hyperedge that contains the vertex. That is, if any hyperedge $E_i \in V_1$ becomes *safe*, then we remove E_i from V_1 and delete all the vertices that are *only* contained in E_i. On the other hand, once a hyperedge in V_1 becomes *dangerous*-2 (it has k_2 vertices, all the *uncolored* vertices in that hyperedge become *trouble-vertex*-2 and we skip the coloring of all such vertices. After the coloring of all vertices in V_1, hyperedges in $\mathcal{E}_1$ are surviving hyperedges. Then we check all the hyperedges in G that are adjacent to the hyperedges in $\mathcal{E}_1$. If any of these hyperedges is not in the *safe* state, then we add it to $\mathcal{E}_1$ and also add all its *uncolored* vertices to V_1. Now we repeat the coloring process described above for these newly added *uncolored* vertices. This exploration of the dependency graph terminates if, either there is no more hyperedge to color, or the number of *surviving*-1 hyperedges in $\mathcal{E}_1$ is

```
Phase-2 Coloring($x$)
Input: a trouble-vertex-1 vertex $x \in V$
Output: a color in {red, blue} or FAIL
1. Start from $\mathcal{E}(x)$ to explore $G$ in order to find the connected
      components of all the surviving-1 hyperedges around $x$
2. If the size of the component is larger than $c_1 \log N$
      Abort and return FAIL
3. Repeat the following until a good coloring is found
      (a) Color all the vertices in $C_1(x)$
      (b) Explore the dependency graph of $G|_{S_1(x)}$
      (c) Check if the coloring is good
4. Return the color of $x$ in the good coloring
```

Figure 4: Phase 2 coloring algorithm

greater than $c_1 \log N$, where c_1 is some absolute constant. The following Lemma shows that, almost surely, the size of $C_1(x)$ is at most $c_1 \log N$.

Lemma 6.2 ([2]). *Let $S \subseteq G$ be the set of surviving hyperedges after the first phase. Then with probability at least $1 - O(\frac{1}{N^2})$ (over the choices of random coloring), all connected components $C_1(x)$ of $G|_S$ have sizes at most $c_1 \log N$.*

Random coloring. Since $C_1(x)$ is not connected to any *surviving*-1 hyperedges in H, we can color the vertices in the connected component $C_1(x)$ without considering any other hyperedges that are outside $C_1(x)$. Now we follow a similar coloring process as in Phase 1 to color the vertices in $C_1(x)$ uniformly at random and in an arbitrary ordering. The only difference is, we ignore all the vertices that are already colored *red* or *blue*, and if $k_1 + k_2$ vertices in a hyperedge get colored monochromatically, and all the rest of vertices in the hyperedge are in *trouble-vertex*-1 state, then this hyperedge will be in *dangerous*-2 state and all the uncolored vertices in it will be in *trouble-vertex*-2 state. Analogously we define *unsafe*-2 hyperedges as hyperedges whose vertices are either colored or in *trouble-vertex*-2 state and all the colored vertices are monochromatic. Finally, we say a hyperedge is a *surviving*-2 edge if it is in either *dangerous*-2 state or *unsafe*-2 state.

Let $S_1(x)$ be the set of surviving hyperedges in $C_1(x)$ after all vertices in $C_1(x)$ are either colored or in *trouble-vertex*-2 state. Now we explore the dependency graph of $S_1(x)$ to find out all the connected components. Another application of Lemma 6.2 to $G|_{S_1(x)}$ shows that with probability at least $1 - O(\frac{1}{\log^2 N})$ (over the choices of random coloring), all connected components in $G|_{S_1(x)}$ have sizes at most $c_2 \log \log N$, where c_2 is some constant. We say a Phase 2 coloring is *good* if this condition is satisfied. Now if a random coloring is not good, then we erase all the coloring performed during Phase 2 and repeat the above coloring and exploring dependency graph process. We keep doing this until we find a good coloring. Therefore, in *expected* $\text{polylog}(N)$ time we can color $C_1(x)$ such that each connected component in $G|_{S_1(x)}$ has size at most $c_2 \log \log N$.

Running time analysis. Combining the analysis above with an argument similar to the running time analysis of Phase 1 coloring gives

Claim 6.3. *Phase 2 coloring can be done in* expected $\text{polylog}(N)$ *time.*

6.5 Phase 3 coloring

In Phase 3, given a vertex x (which is necessarily *trouble-vertex*-2), we grow a connected component which includes x as in Phase 2, but of *surviving*-2 hyperedges. Denote this connected component of *surviving*-2 hyperedges by $C_2(x)$. By our Phase 2 coloring, the size of $C_2(x)$ is no greater than $c_2 \log \log N$. We then color the vertices in this connected component by exhaustive search. The existence of such a coloring is guaranteed by the Lovász Local Lemma (Lemma 3.1).

Claim 6.4. *The time complexity of Phase 3 coloring is at most* $\text{polylog}(N)$.

Proof. Using the same analysis as for Phase 2, in time $O(\log \log N)$ we can explore the dependency graph to grow our connected component of *surviving*-2 hyperedges. Exhaustive search of a valid two-coloring of all the vertices in $C_2(x)$ takes time at most $2^{O(|C_2(x)|)} = 2^{O(\log \log N)} = \text{polylog}(N)$, as $|C_2(x)| \le c_2 \log \log N$ and each hyperedge contains k vertices. □

Finally, we remark that using the same techniques as those in [2], we can make our local computation algorithm

```
Phase-3 Coloring($x$)
Input: a trouble-vertex-2 vertex $x \in V$
Output: a color in {red, blue}
1. Start from $\mathcal{E}(x)$ to explore $G$ in order to find the connected
       component of all the surviving-2 hyperedges around $x$
2. Go over all possible colorings of the connected component
       and color it using a feasible coloring.
3. Return the color $c$ of $x$ in this coloring.
```

Figure 5: Phase 3 coloring algorithm

run in parallel and find an ℓ-coloring of a hypergraph for any $\ell \geq 2$ (an ℓ-coloring of a hypergraph is to color each vertex in one of the ℓ colors such that each color appears in every hyperedge).

7 k-CNF

As another example, we show our hypergraph coloring algorithm can be easily modified to compute a satisfying assignment of a k-CNF formula, provided that the latter satisfies some special properties.

Let H be a k-CNF formula on m Boolean variables $x_1, \ldots, x_m$. Suppose H has N clauses $H = A_1 \wedge \cdots \wedge A_N$ and each clause consists of exactly k distinct literals.[6] We say two clauses A_i and A_j *intersect* with each other if they share some variable (or the negation of that variable). As in the case for hypergraph coloring, k and d are fixed constants and all asymptotics are with respect to the number of clauses N (and hence m, since $m \leq kN$). Our main result is the following.

Theorem 7.1. *Let H be a k-CNF formula with $k \geq 2$. If each clause intersects no more than d other clauses and furthermore k and d are such that there exist three positive integers k_1, k_2 and k_3 satisfying the followings relations:*

$$\begin{aligned} k_1 + k_2 + k_3 &= k, \\ 8d(d-1)^3(d+1) &< 2^{k_1}, \\ 8d(d-1)^3(d+1) &< 2^{k_2}, \\ e(d+1) &< 2^{k_3}, \end{aligned}$$

then there exists a local computation algorithm that, given any sequence of queries to the truth assignments of variables $(x_1, x_2, \ldots, x_s)$, with probability at least $1 - 1/N^2$, returns a consistent truth assignment for all x_i's which agrees with some satisfying assignment of the k-CNF formula H. Moreover, the algorithm answers each single query in expected $O((\log N)^c)$ *time, where c is some constant (depending only on k and d).*

[6] Our algorithm works for the case that each clause has at least k literals; for simplicity, we assume that all clauses have uniform size.

Proof [Sketch]: We follow a similar algorithm to that of hypergraph two-coloring as presented in Section 6. Every clause will be in either *initial*, *safe*, *unsafe*-1, *unsafe*-2, *dangerous*-1 or *dangerous*-2 state. Every variable will be in either *unassigned*, *true*-1, *false*-1, *trouble-variable*-1 or *trouble-variable*-2 state. Initially every clause is in *initial* state and every variable is in *unassigned* state. Suppose we are asked about the value of a variable x_i. If x_i is in *initial* state, we randomly choose from $\{\mathbf{true}, \mathbf{false}\}$ with equal probabilities and assign it to x_i. Then we update all the clauses that contain either x_i or $\bar{x}_i$ accordingly: If the clause is already evaluated to $\mathbf{true}$ by this assignment of x_i, then we mark the literal as *safe*; if the clause is in *initial* state and is not *safe* yet and x_i is the k_1^{th} literal in the clause that has been assigned values, then the clause is marked as *dangerous*-1 and all the remaining unassigned variables in that clause are now in *trouble*-1 state. We perform similar operations for clauses in other states as we do for the hypergraph coloring algorithm. The only difference is now we have $\Pr[A_i \text{ becomes } \textit{dangerous}\text{-}1] = 2^{-k_1}$, instead of 2^{1-k_1} as in the hypergraph coloring case. Following the same analysis, almost surely, all connected components in the dependency graph of *unsafe*-1 clauses are of size at most $O(\log N)$ and almost surely all connected components in the dependency graph of *unsafe*-2 clauses are of size at most $O(\log \log N)$, which enables us to do exhaustive search to find a satisfying assignment. $\square$

8 Concluding Remarks and Open Problems

In this paper we propose a model of local computation algorithms and give a technique which can be applied to construct polylogarithmic time local computation algorithms. It would be interesting to understand the scope of problems which can be solved with such algorithms and to develop other techniques that would apply in this setting.

Acknowledgments

We would like to thank Ran Canetti, Tali Kaufman, Krzysztof Onak and the anonymous referees for useful discussions and suggestions.

References

[1] N. Ailon, B. Chazelle, S. Comandur, and D. Liu. Property-preserving data reconstruction. *Algorithmica*, 51(2):160–182, 2008.

[2] N. Alon. A parallel algorithmic version of the Local Lemma. *Random Structures and Algorithms*, 2:367–378, 1991.

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

[4] N. Alon, A. Bar-Noy, N. Linial, and D. Peleg. On the complexity of radio communication. In *Proc. 21st Annual ACM Symposium on the Theory of Computing*, pages 274–285, 1989.

[5] N. Alon, A. Bar-Noy, N. Linial, and D. Peleg. Single round simulation on radio networks. *Journal of Algorithms*, 13:188–210, 1992.

[6] R. Andersen, C. Borgs, J. Chayes, J. Hopcroft, V. Mirrokni, and S. Teng. Local computation of pagerank contributions. *Internet Mathematics*, 5(1–2):23–45, 2008.

[7] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *Proc. 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 475–486, 2006.

[8] J. Beck. An algorithmic approach to the Lovász Local Lemma. *Random Structures and Algorithms*, 2:343–365, 1991.

[9] P. Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, 2006.

[10] B. Chazelle and C. Seshadhri. Online geometric reconstruction. In *SoCG*, pages 386 – 394, 2006.

[11] S. Cook. A taxonomy of problem with fast parallel algorithms. *Information and Control*, 64(1–3):2–22, 1985.

[12] P. Erdös and L. Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In A. Hajnal et. al., editor, *"Infinite and Finite Sets", Colloq. Math. Soc. J. Bolyai*, volume 11, pages 609–627. North-Holland, 1975.

[13] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, 45:653–750, 1998.

[14] A. V. Goldberg, S. A. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1988.

[15] A. Hasidim, J. Kelner, H. N. Nguyen, and K. Onak. Local graph partitions for approximation and testing. In *Proc. 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 22–31, 2009.

[16] G. Jeh and J. Widom. Scaling personalized web search. In *Proceedings of the 12th International Conference on World Wide Web*, pages 271–279, 2003.

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

[18] J. Katz and L. Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *Proc. 32nd Annual ACM Symposium on the Theory of Computing*, pages 80–86, 2000.

[19] F. Kuhn. Local multicoloring algorithms: Computing a nearly-optimal tdma schedule in constant time. In *STACS*, pages 613–624, 2009.

[20] F. Kuhn and T. Moscibroda. Distributed approximation of capacitated dominating sets. In *SPAA*, pages 161–170, 2007.

[21] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer. Fast deterministic distributed maximal independent set computation on growth-bounded graphs. In *DISC*, pages 273–287, 2005.

[22] F. Kuhn, T. Moscibroda, and R. Wattenhofer. What cannot be computed locally! In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*, pages 300–309, 2004.

[23] F. Kuhn, T. Moscibroda, and R. Wattenhofer. The price of being near-sighted. In *Proc. 17th ACM-SIAM Symposium on Discrete Algorithms*, pages 980–989, 2006.

[24] F. Kuhn and R. Wattenhofer. On the complexity of distributed graph coloring. In *Proc. 25th ACM Symposium on Principles of Distributed Computing*, pages 7–15, 2006.

[25] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986. Earlier version in STOC'85.

[26] S. Marko and D. Ron. Distance approximation in bounded-degree and general sparse graphs. In *APPROX-RANDOM'06*, pages 475–486, 2006.

[27] A. Mayer, S. Naor, and L. Stockmeyer. Local computations on static and dynamic graphs. In *Proceedings of the 3rd Israel Symposium on Theory and Computing Systems (ISTCS)*, 1995.

[28] R. Moser. A constructive proof of the general Lovász local lemma. In *Proc. 41st Annual ACM Symposium on the Theory of Computing*, pages 343–350, 2009.

[29] R. Moser and G. Tardos. A constructive proof of the general Lovász local lemma. *Journal of the ACM*, 57(2):Article No. 11, 2010.

[30] M. Naor and L. Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.

[31] H. N. Nguyen and K. Onak. Constant-time approximation algorithms via local improvements. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 327–336, 2008.

[32] M. Parnas and D. Ron. Approximating the minimum vertex cover in sublinear time and a connection to distributed

algorithms. *Theoretical Computer Science*, 381(1–3):183–196, 2007.

[33] M. E. Saks and C. Seshadhri. Local monotonicity reconstruction. *SIAM Journal on Computing*, 39(7):2897–2926, 2010.

[34] T. Sarlos, A. Benczur, K. Csalogany, D. Fogaras, and B. Racz. To randomize or not to randomize: Space optimal summaries for hyperlink analysis. In *Proceedings of the 15th International Conference on World Wide Web*, pages 297–306, 2006.

[35] D. Spielman and S. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proc. 36th Annual ACM Symposium on the Theory of Computing*, pages 81–90, 2004.

[36] Y. Yoshida, Y. Yamamoto, and H. Ito. An improved constant-time approximation algorithm for maximum matchings. In *Proc. 41st Annual ACM Symposium on the Theory of Computing*, pages 225–234, 2009.