Local Computations on Static and Dynamic Graphs[*]

Alain Mayer[†] Moni Naor[‡] Larry Stockmeyer[§]

Abstract. The purpose of this paper is a study of computation that can be done *locally* in a *dynamic* distributed network. By locally we mean within time (or distance) independent of the size of the network and by dynamic we mean that the underlying graph is not stable and links continuously fail and come-up. One of the main contributions of this work is a definition of *robustness*, which captures the nature of an algorithm performing well in such an environment.

The second important contribution is the *happy coloring & orientation* tool and abstraction and its application to robust and local solutions to the weak coloring and (d, m)-dining philosophers problems of Naor and Stockmeyer [28]. The latter problem is similar to the usual dining philosophers problem, except that each philosopher has access to d forks but needs only m of them to eat. We give a robust local solution if $m \le \lceil d/2 \rceil$ (necessity of this inequality for any local solution was known previously).

In the same environment we also investigate (1) the amount of initial symmetry-breaking needed to solve certain problems locally (for example, our algorithms need considerably less symmetry-breaking than having a unique ID on each node), and (2) the single-step color reduction problem: given a coloring with c colors of the nodes of a graph, what is the smallest number of colors c' such that every node can recolor itself with one of c' colors as a function of its immediate neighborhood only.

Key words. distributed computation, local computation, graph labeling problem, resource allocation, dining philosophers problem

AMS subject classifications. 68M10, 68Q20, 68Q22, 68R05, 68R10

[*]Preliminary version appeared in Proc. 3rd Israeli Symp. on Theory of Computing and Systems, 1995, pp. 268–278.

[†]Dept. of Computer Science, Columbia University, New York, NY 10027, USA. E-mail: mayer@cs.columbia.edu. Part of this work was done while the author was visiting the IBM T. J. Watson Research Center. Partially supported by NSF grant CCR-93-16209 and CISE Institutional Infrastructure Grant CDA-90-24735

[‡]Incumbent of the Morris and Rose Goldman Career Development Chair, Dept. of Applied Mathematics and Computer Science, Weizmann Institute, Rehovot 76100, Israel. E-mail: naor@wisdom.weizmann.ac.il. Part of this work was done while at the IBM Almaden Research Center. Research supported by BSF Grant 94-00032/1 and by an Israel-France grant.

[§]IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA. E-mail: stock@almaden.ibm.com.

1 Introduction

Locality is an important feature of a distributed system. Each processor is directly connected to at most some fixed number of others. In such an environment it is easy to see that in t time-units, a processor can only learn about the situation at processors which are within distance at most t from itself. However, typical problems arising in a distributed system require that the values computed at different processors must fit together in some global way. This observation gives rise to numerous questions of how much time (and hence how much a processor needs to know of its neighborhood) is required to solve a given problem.

Unreliability is an inherent problem of a distributed system. Connections between processors can fail (and later recover) and processors might crash. Hence a desirable feature of a computation in a distributed system is fault-tolerance. Up to now most of the work for changing (dynamic) networks used the assumption that the sequence of topological changes is finite. This assumption is called *eventual quiescence* (see, for instance, the fundamental work of Afek et al. in [1]). It has been recognized that eventual quiescence is a strong assumption for large networks and difficult to ensure in practical systems (see the work of Awerbuch et al. in [8]).

It should come as no surprise that local algorithms can be made fault-tolerant under a weaker assumption than eventual quiescence. While it has been recognized that locality is a useful tool for fault-tolerance (see Section 1.1), there has been no formal quantifiable definition of locality, on which fault-tolerant algorithms can be built. An important contribution of this work is the introduction of a new, quantifiable assumption, weaker than eventual quiescence, on the condition of allowable topological changes in the network and a corresponding definition for what it means for an algorithm to be correct. This new definition, which we call *robustness*, is needed, since our assumption allows, from a global perspective, perpetual changes in the network.

Informally, an algorithm is robust if, for every node v, it operates correctly at v provided that a sufficiently large neighborhood around v remains stable for a sufficiently long time. We are especially interested in the case that the radius of the neighborhood and the required stability time are constant independent of the size of the network.

Given a particular computational model, certain problems are more basic than others as tools for the solution of other problems. Our second important contribution, presented in Section 3, is the *happy coloring & orientation* tool and abstraction. This labeling is a weakened version of a *happy partition* (see [21]). We show a robust local algorithm to compute it. The algorithm is then used in later sections as a basic building block.

The problem of an ever changing network is even more severe in reactive problems, i.e., problems which run "forever" and the input at every processor/node changes in time, such as mutual exclusion, dining philosophers, etc. In Section 4, we present a local algorithm to the *(d, m)-Dining Philosophers Problem* (*(d, m)-DPP*) introduced in [28], a variant of the well-known dining philosophers problem (see, e.g., [11, 23, 31]), where a processor has access to d forks, but only needs m of them in order to eat. Based on a happy coloring & orientation of the conflict graph, we show how to solve the problem locally and robustly for $m \leq \lceil d/2 \rceil$. (It has been shown in [28], that for $m > \lceil d/2 \rceil$, there is no local algorithm for (d, m)-DPP. Moreover, this holds even when the network and conflict graph are static.) Our algorithm employs the "doorway algorithm" from Choy and Singh [12]. Given our discussion about fault-tolerance, we can show that we need only a constant-size neighborhood around a node to be stable for a constant period of time in order for that node to progress, i.e., if it was hungry at the start of the period, it will obtain enough forks to begin eating by the end of the period. The results about (d,m)-DPP described above concern the case that each resource (fork) is shared by two processors. In Section 5 the results are extended to the case where each resource can be shared by up to r processors. In this case, a necessary and sufficient condition for a local solution is $m \leq \lceil d/r \rceil$ in both the static and dynamic case.

We continue (Section 6) by showing that a happy coloring & orientation can be used to give a simpler local algorithm than in [28] to obtain a *weak coloring* (a coloring in which every node has at least one neighbor colored differently). In Section 7 we consider the question of how much initial symmetry breaking in the network is needed to solve certain problems locally. For the results mentioned above, we need only that each node have a local ordering of its incident links and that each link have an initial orientation. (In the case of DPP with resource sharing greater than 2, the algorithm needs an "orientation" of each resource, i.e., a total order on the processors that share the resource.) This is weaker than the usual assumption of unique ID's on the nodes. We show, however, that the weaker symmetry-breaking assumption cannot be used in place of unique ID's in general. Finally, in Section 8 we investigate the *color reduction problem*: starting with a legal coloring with c colors what is the smallest number of colors c' such that every vertex can recolor itself legally with one of c' colors as a function of its *immediate* neighborhood only. In particular for graphs with degree bounded by some constant, the best known bound on c' is $c' = O(\log \log c)$, obtained by a nonconstructive solution of Szegedy and Vishwanathan [32]. By applying a construction of certain set systems in [25, 27], we obtain a constructive color reduction method, closely related to the one of [32], also having $c' = O(\log \log c)$. (The solutions are not restricted to constant-degree graphs. In Section 8 we discuss how c' depends on both c and the degree.)

1.1 Related work

We now survey several lines of research that are related to the issue of locality and its use in fault-tolerance:

- Quality of Local Decisions in a Dynamic Network: An early treatment can be found in Awerbuch and Even [7]: they introduce the notion of *eventual connectivity* (roughly, no edge-cut persists indefinitely) and notice that the difficulty of designing a routing algorithm under this assumption is that completely local decisions are either insufficient to ensure correctness or lead to inefficiency. Similar considerations for the problem of end-to-end communication are made in Afek and Gafni [2] and many other papers on this subject.

- Self-Stabilization: Both in our model and in the model of *self-stabilization* (see Schneider [30] for a survey) the goal is to reach a legal state when started in an unknown state and to try to regain a legal state when a change (e.g., failure) occurs. E.g., the problem might be, starting from an arbitrary node coloring, to find a legal node coloring. The works of Afek, Kutten, and Yung [3] and Awerbuch, Patt-Shamir, and

Varghese [9] introduce the notion of *locally checkable* problems. These are problems whose legal states can be expressed as a conjunction of local (node-)predicates. However, to compute a locally checkable problem (to bring it to a legal state) might still need time (convergence-time) which depends on the size of the network. Finally, a self-stabilizing algorithm must overcome even maliciously chosen initial states (e.g., arbitrary messages in transit).

- Local-Global Phenomena: Linial [21] surveys many aspects of the question of which global graph parameters are determined by local properties. In particular, Linial et al. [22] considered the problem of inferring the average of a non-negative function defined on the nodes of a graph. They showed that if there is a lower bound μ on the value of the function in *all* neighborhoods of distances up to r then the average of the function value must be at least $\mu n^{-O(1/\log r)}$ and that this result is tight.

- Local Mending: Recently Kutten and Peleg [18, 19] considered the notion of *fault-local mending.* This is a way to measure the complexity of algorithms for labeling problems (e.g., they consider "maximal independent set") where the algorithm should be efficient as a function of the number of faults (and not the size of the network). In comparison with our approach, the faults are a one-time event, i.e. they assume global quiescence. On the other hand, any labeling problem that has a local robust solution in our scenario can be locally mended.

2 Definitions

We first give some (informal) definitions concerning graphs. All graphs in this paper are simple and undirected. Part of the structure of a graph is a *port labeling*, i.e., a labeling of the vertex-edge incidences with labels from some finite totally-ordered set, such that the incidences at a given vertex receive distinct labels. Thus, for example, after receiving a message on link e, the algorithm may send a message back on link e. We consider only networks of bounded degree. We assume that initially each link is oriented and that both endpoints agree on this orientation. This is obviously a weaker assumption than the usual assumption of unique ID's. It turns out that this assumption is sufficient for our purposes and it is well suited since a new node joining the network only needs to arrange itself with its immediate neighbors rather than with all other nodes in the network. For a graph $G = (V, E)$ and vertices $v, w \in V$, let $dist_G(v, w)$ be the distance (length of a shortest path) in G from v to w. For a vertex v and a nonnegative integer r, let $B_G(v, r)$ denote the *ball* around vertex v of radius r, i.e., the set of vertices w such that $dist_G(v, w) \leq r$. A *centered graph* is a pair (H, s) where H is a graph and s is a vertex of H. The *radius* of (H, s) is the maximum distance from s to any vertex or edge of H.

In a network of dynamic topology (see e.g., [1]), each link may fail and recover an unbounded number of times. Nodes do not fail, but we can model a node failure by failure of all its incident links. Link failure is simply the loss of the connection between two endpoints which may result in a loss of a message. Nodes can detect failure and recovery of an incident link. For resource-allocation problems, there is a given *conflict graph* where each node represents a processor and each edge represents a resource (a "fork") which is shared by the two endpoints. We assume that the conflict graph is a subgraph of the network (as is usually done, see, e.g., [6]). Hence a link failure equals loss of that particular resource. As opposed to the network-graph, node failures here are separate and represent failures of processors.

We now recall the notion of a "locally checkable labeling" (LCL) from [28], slightly modified for our purposes. An *LCL* $\mathcal{L}$ consists of a positive integer r (called the *radius* of $\mathcal{L}$), a finite set Σ of *input labels*, a finite set Γ of *output labels*, and a finite set $\mathcal{C}$ of *locally consistent labelings*. Each element of $\mathcal{C}$ is a centered graph of radius at most r where each vertex is labeled with a member of Γ. Given a graph $G = (V, E)$ and a labeling $\lambda : V \to \Gamma$, the labeling λ is *$\mathcal{L}$-legal* if, for every $v \in V$, there is a $(H, s) \in \mathcal{C}$ and an isomorphism π mapping $B_G(v, r)$ to H such that $\pi(v) = s$ and such that π respects the port labeling and the vertex labeling λ. Essentially, the set $\mathcal{C}$ gives a "truth table" of all locally consistent labelings. For the labeling problems considered in this paper, the legality of the output labeling does not depend on the input labeling. However, the input labeling can be used by an algorithm to help it find an output labeling (e.g., for "symmetry breaking"). Since a graph is supplied with a port labeling, an edge labeling (e.g., an edge orientation) can be encoded in vertex labels. The correctness of an algorithm for the particular reactive problem (d, m)-DPP is discussed at the beginning of Section 4.

The above definition was given for a *static* graph G, i.e., the topology of G remains fixed over time. But note that $\mathcal{L}$-legality is essentially one big conjunction of predicates over local neighborhoods of radius r. Thus changes in the topology of G only affect the legality of labelings within distance r of that change. Hence we can define legality in a dynamic graph as follows. We say that a ball $B(v, r)$ is *stable* (for a time t) if there are no topological changes in $B(v, r)$ (for time t) and the input labeling in $B(v, r)$ does not change (for time t).

Definition 2.1 *We say that a labeling algorithm is (r, t)-robust if (1) it produces a legal (according to LCL) output labeling on vertex v whenever $B(v, r)$ is stable for time t, and (2) it does not change this legal labeling on vertex v as long as $B(v, r)$ is stable thereafter.*

For *reactive* algorithms we use the following definition:

Definition 2.2 *We say that a reactive algorithm is (r, t)-robust if it (1) produces* progress *on vertex v whenever $B(v, r)$ is stable for time t, and (2) maintains* safety *properties at all times regardless of changes in topology or input labeling.*

For example, in the case of resource allocation problems, such as the dining philosophers problem and the variant (d, m)-DPP, progress means that if v becomes hungry and if $B(v, r)$ is stable for time t, then v will obtain enough resources to start eating within time t. Safety means that two processors never own the same resource at the same time. In our solution safety is achieved by having each process restart its DPP algorithm upon detection of instability.

The *composition* $P_1 \circ P_2$ of two programs is formed by running P_1 and P_2 in parallel, with the output labeling of P_1 used as the input labeling of P_2. For the compositions in this paper, P_1 is always a labeling algorithm; P_2 is either a labeling algorithm or a resource allocation algorithm.

Theorem 2.1 *If P_1 is a (r_1, t_1)-robust algorithm and P_2 is a (r_2, t_2)-robust algorithm, then the composition $P_1 \circ P_2$ is a $(r_1 + r_2,\ t_1 + t_2)$-robust algorithm.*

Proof. Note that for each vertex w in $B(v, r_2)$ (including v itself), $B(w, r_1)$ is stable for time t_1, and hence P_1 produces a correct output on w. Now stability of each such $B(w, r_1)$ is guaranteed for an additional time t_2, and hence P_1 maintains a stable output in $B(v, r_2)$, so P_2 is guaranteed to produce a correct output (or behavior) on v. $\square$

For particular algorithms P_1 and P_2, it might be possible to obtain better bounds on the robustness of their composition than the general bound given in this theorem.

Since the focus of our investigation is "constant-time" algorithms, we say that an algorithm is "local" if it is (r,t)-*robust* for both r, t being constants independent of the size of the network. Note that a (r,t)-*robust* solution to the (d,m)-DPP implies that the response time is bounded by t and that the maximum length of a waiting chain is r. Lynch [23] was the first to explicitly consider the response time for the dining philosophers problem. The goal of successive works was to make the response time of a node to depend only on its local neighborhood in the conflict-graph (see, e.g., [12] and [10]). The importance of short waiting chains has been pointed out in [12], [28], and [6] (where this property is called r-wait-freedom).

Our computational model is an asynchronous message passing system with an upper bound ν on the time for a message to traverse a single link, and ν also includes the time for local computation at a processor. Thus, time bounds are given as functions of ν.

3 Happy Coloring & Orientation

In this section we first define a *happy coloring & orientation* of a graph and then show a one-step local algorithm to compute it.

The problem of finding *happy partitions* is well known: A partition of the vertex set of a graph $V = A \dot{\bigcup} B$ is called *happy* if every $x \in A$ has at least half of its neighbors in B, and vice versa. Such partitions always exist and a sequential algorithm for finding such a partition can be obtained by considering local improvements: if a there is a node that is unhappy, flip its part; the number of edges between A and B only increases by the process. However, the problem cannot be solved locally, and furthermore, Linial and Saks conjecture a $\Omega(\sqrt{n})$ time lower bound for any distributed algorithm (see [21] and see [29] for the parallel complexity of the problem). Therefore we turn now to a weaker form of a partition: We are given a graph which has an arbitrary initial orientation on its edges and an *arbitrary* initial coloring of its vertices with two colors (say a and b). A happy coloring & orientation is a 2-coloring (i.e., a partition) and an orientation of the given graph with the following properties at each vertex v, where d_v is the degree of v, and "indegree$_v$ from c" is the number of neighbors w of v such that w is colored c and the edge (w, v) is directed towards v:

- $(\text{color}_v = a) \Rightarrow (\text{indegree}_v \text{ from } a + \text{neighbors with } b) \geq \lceil d_v/2 \rceil$
- $(\text{color}_v = b) \Rightarrow (\text{indegree}_v \text{ from } b + \text{neighbors with } a) \geq \lceil d_v/2 \rceil$

```
REPEAT FOREVER
    IF (color$_v$ $= b$) AND ((indegree$_v$ from $b$ + neighbors with $a$) $< \lceil d_v/2 \rceil$)
        reverse all incoming edges;
        color$_v$ $:= a$;
        send "my color changed to $a$" to all neighbors;
    ELSIF (color$_v$ $= a$) AND ((indegree$_v$ from $a$ + neighbors with $b$) $< \lceil d_v/2 \rceil$)
        reverse all incoming edges;
        color$_v$ $:= b$;
        send "my color changed to $b$" to all neighbors;
END
```

Figure 1: Happy coloring & orientation

In analogy to a happy partition we can describe the above as follows: A partition of the vertex set of a graph $V = A \dot{\bigcup} B$ and an orientation of the edges is called a happy coloring & orientation if every $x \in A$ has at least half of its neighbors either in B or pointing towards x, and vice versa. Figure 1 shows an algorithm to compute a happy coloring & orientation. In addition, whenever a new link incident to v comes up (including the start of the algorithm when all links come up), v sends its current color to the vertex at the other end of the link. Whenever a new link comes up, both endpoints are given the initial orientation of the link. Recall that the orientation of each edge is encoded in the vertex labels of the endpoints of the edge; one of the requirements for a labeling to be legal is that the endpoints agree on the orientation.

Theorem 3.1 *The algorithm in Figure 1 is a $(2, 2\nu)$-robust algorithm for a happy coloring & orientation.*

Proof. First note that a vertex which has a legal happy coloring & orientation does not have to change its color as a result of a color change by a neighbor. This is because a changing neighbor reverses all its incoming edges and thus contributes to the indegree of any of its neighbors. Suppose that $B(v, 2)$ is stable for time 2ν, starting at time t_0. By time $t_0 + \nu$, v will learn, for each of its neighbors w, a color c_w that w had at some time $t_w \geq t_0$. Suppose for the moment that no w changes this color c_w. Now v changes its color from b to a only if its outdegree to b-colored vertices is at least $\lceil d_v/2 \rceil$. The new color a is legal because v has at least $\lceil d_v/2 \rceil$ neighbors colored b. If on the other hand any of these neighbors changes from b to a then, as noted above, they will contribute to the indegree of v and thus the new color a is legal as well. A neighbor changes color no later than time $t_0 + \nu$, so v learns of any changed orientations no later than time $t_0 + 2\nu$. The case of v changing from a to b is symmetric. Furthermore, topological changes at a distance 3 from a vertex v will not cause any neighbor of v to change color, so such changes cannot affect the color of v or the orientation of any of its links. $\square$

4 (d, m)-Dining Philosophers Problem

Recall the definition of the (d, m)-Dining Philosophers Problem: There is a given *conflict graph* of minimum degree d where each vertex represents a processor and each edge represents a resource (a "fork") which is shared by the two endpoints. We assume that the conflict graph is a subgraph of the network (as is usually done, see, e.g., [6]). Hence a link failure equals loss of that particular resource. Vertex failures are separate and represent failures of processors.

At any time, a fork can be "owned" by at most one of the processors which share it. Each processor can be in one of three states: resting, hungry, or eating. A resting processor can become hungry at any time. In order to eat, a processor must obtain at least m forks. A processor eats for at most a bounded time (denoted τ), after which it returns to the resting state. A processor p can attempt to "grab" a certain fork, and to release an owned fork. The grab operation will fail if the fork is currently owned by the other processor q; if this occurs, p may decide to wait for q to release the fork. Communication is via message-passing, where we assume ν to be an upper bound for message propagation between any two adjacent vertices. We require a "local" solution, i.e., (r, t)-robust, where r, t are independent of the size of the network. Such a solution implies that any initial coloring algorithm used must also be (r, t)-robust, that the *response time* is bounded by t, and that the maximum length of a *waiting chain* which can develop is bounded by r. As pointed out by Choy and Singh [12], a difficulty with long waiting chains is that if a processor p fails while holding a fork, the failure will affect every processor behind p in the waiting chain.

We now turn to the design of a local algorithm that solves the (d, m)-DPP problem for any bounded-degree conflict graph with $m \leq \lceil d/2 \rceil$. (As already noted, this inequality is necessary for any local solution [28].) The basic idea of the algorithm is to use a happy coloring & orientation algorithm to reduce the problem to a standard dining philosophers problem on a subgraph of the conflict graph. The coloring and subgraph have the property that if two vertices conflict (i.e., are contesting for the same resource in the subgraph) then they are colored differently. We can then use an algorithm described by Choy and Singh [12], which is local provided that such a coloring is given. (It is well-known that such colorings cannot be found locally if we have to use the entire conflict graph rather than a subgraph of it [20]).

We make use of the happy coloring & orientation as follows. Forks on monochrome edges are statically allocated according to the orientation. I.e., if edge $e = (u, v)$ is oriented from u to v and both u and v are colored with the same color, then v gets e "permanently" (as long as the happy coloring & orientation does not change as a result of topological updates). Thus if we delete these edges from our conflict graph we obtain a bipartite graph. The edges in this bipartite graph represent the forks that are contested. Note that since any vertex v needs exactly m forks, it only needs to consider $m-$(number of statically allocated forks to v) in this bipartite graph. So for analysis purposes we might just as well assume that the degree of the bipartite graph is bounded by m. We say that two vertices are *competitors* if they are connected by an edge in this bipartite graph. By construction, if two vertices are competitors then they are colored differently.

If a vertex becomes hungry it will run the following two-phase algorithm on this bipartite graph: In the first phase the vertex tries to enter a single doorway and in the second phase

```
Phase 1: To enter a doorway
Send(req-doorway) to all competitors;
Wait(ack-doorway) from all competitors;

On receiving req-doorway from vertex $w$:
Wait(outside-doorway);
Send(ack-doorway) to $w$;

Phase 2: To gather forks inside a doorway
Send(req-fork) to any competitor $w$ with $\textit{have-fork}_v(w) = 0$;
If receive ack-fork from $w$, set $\textit{have-fork}_v(w) := 1$;
Wait $\textit{have-fork}_v(w) = 1$ for all competitors $w$;

On receiving a req-fork from vertex $w$:
Wait(not-eating);
Wait(outside-doorway or $w$ has better color);
Wait($\textit{have-fork}_v(w) = 1$);
Send(ack-fork) to $w$ and set $\textit{have-fork}_v(w) := 0$;
If inside a doorway, Send(req-fork) to $w$;
```

Figure 2: The doorway algorithm at vertex v

the vertex collects forks in a preemptive manner from all its competitors. Priority is given according to the colors. This doorway algorithm is from Choy and Singh [12].

A single doorway works as follows: To enter the doorway, a processor signals all its competitors and waits for acknowledgements. On receiving all the acknowledgements, the processor enters the doorway. A processor which has not entered the doorway will acknowledge its competitors when it receives the signals. But processors that have entered the doorway do not acknowledge until they have finished eating. As soon as a processor is in the doorway it starts to gather the forks as follows: It requests the forks from all its competitors and waits for them. On receiving all forks, the processor starts eating. A processor will send the fork to a requesting competitor if it is not eating and if it either has a lower-priority color or if it is outside the doorway. A more formal description can be found in Figure 2. In this description, each vertex v has a variable $\textit{have-fork}_v(w)$ for each competitor w, which is 1 if v has the fork it shares with w, or 0 if v does not have the fork. When the edge (v, w) first appears, exactly one of $\textit{have-fork}_v(w)$ and $\textit{have-fork}_w(v)$ is 1 (depending, say, on the initial orientation of the edge). The algorithm maintains the *safety* property that, at any time, at most one of $\textit{have-fork}_v(w)$ and $\textit{have-fork}_w(v)$ is 1.

We assume that there is an underlying mechanism to detect topological changes. The algorithm running at vertex v is restarted whenever the underlying happy coloring & orientation changes in $B(v, 1)$ (as a result of a topological change). We get the following:

Theorem 4.1 *For $m \leq \lceil d/2 \rceil$ the composition of the algorithm in Figure 1 with the algorithm in Figure 2 is a $(5,\ (2m+2)\tau + (3m+8)\nu)$-robust algorithm for (d, m)-DPP.*

Proof. We first prove the response time to be $(2m + 2)\tau + (3m + 6)\nu$. Let $T_{c,p}$ denote the upper bound on the time it takes for a vertex with color c ($c \in \{a, b\}$) to finish phase p ($p \in \{1, 2\}$). Assume that color a has priority over color b. First, $T_{a,2} = \tau + 2\nu$: a vertex with higher priority simply grabs the forks from all its non-eating competitors. Second, $T_{b,2} = m(T_{a,2} + \tau + \nu) + 2\nu$. The worst case for this bound occurs at a vertex v colored b that has m competitors all colored a. Suppose that v begins Phase 2 (enters the doorway) at time t. Note that after time t, each competitor can enter the doorway at most once before v gathers enough forks to start eating. This is true because v will not acknowledge any new requests for the doorway until after it eats. Moreover, v will receive an ack-fork from competitor w within time $\delta = T_{a,2} + \tau + \nu$ after w enters the doorway. Therefore, the time for v to gather all its forks is at most m delays of δ plus the initial round-trip time of 2ν for v's first req-fork messages (sent at time t) to be delivered and any ack-fork messages to be returned to v. (This upper bound can occur in the case that, at the same time that v receives an ack-fork from one competitor w, another competitor w' enters the doorway and sends a req-fork to v which arrives quickly (delay 0). Even though w' got v's ack-doorway before v entered the doorway, it could happen that w' just "now" got *all* its ack-doorway's and hence enters the doorway.) For Phase 1 we have $T_{a,1} = T_{b,2} + \tau + 2\nu$ and $T_{b,1} = T_{a,2} + \tau + 2\nu$: a vertex outside the doorway gets inside once all competitors (of opposite color) have finished gathering their forks and eating. Adding the bounds for both phases yields $(2m + 2)\tau + (3m + 6)\nu$ for each color.

Note that the length of the longest waiting chain inside the doorway is at most two (by priority among colors). Thus adding one on account of waiting to get into the doorway yields 3.

The above together with Theorem 3.1 and Theorem 2.1 yields the result. □

5 Resource Sharing Greater than 2

By the result in the previous section and a result in [28], (d, m)-DPP can be solved locally iff $m \leq \lceil d/2 \rceil$, in the case that each resource is shared by two processors. In this section we extend these results to allow each resource to be shared by up to r processors, for some constant r. For the standard dining philosophers problem, where a processor needs all incident resources to eat, the case $r > 2$ easily reduces to the case $r = 2$: if a set N of processors all share some resource, then the conflict graph contains a clique on N. It is not clear how to make this reduction work for (d, m)-DPP. In fact, we find that the condition under which a local solution is possible depends on r: the condition is $m \leq \lceil d/r \rceil$. The algorithm follows the same outline as the algorithm in the previous section. We first find a coloring, and then use this coloring to solve a standard dining philosophers problem using the doorway algorithm. As before, the combined algorithm is robust in a dynamic environment (although with a larger response time and longer waiting chains). The algorithm needs only local symmetry-breaking information. The proof of impossibility for $m > \lceil d/r \rceil$ is similar to the proof in [28] for the case $r = 2$. As in [28], the impossibility result holds even in a static synchronous system, and even with unique ID's on the processors.

```
REPEAT FOREVER
$R(v,j) = \{\, \rho \mid v$ is $j$th in the ordering at resource $\rho \,\}$;
$color(v) =$ smallest $j$ such that $|R(v,j)| \geq m$;
mark 1 on $m$ of the edges to resources in $R(v, color(v))$;
mark 0 on the other edges incident on $v$;
inform all neighbors of this edge marking;
END
```

Figure 3: Happy coloring & marking at vertex v

5.1 The algorithm

It is useful to view the processor-resource incidence information as a bipartite graph, with the processors on one side, the resources on the other, and an edge between processor v and resource ρ if v is one of the processors that share ρ. So each processor has degree at least d and each resource has degree at most r in this graph. (This graph should not be confused with the bipartite graph used in the doorway algorithm in Section 4.) For symmetry breaking, we assume that each processor has a total order on the resources to which it is connected, and each resource has a total order on the processors to which it is connected. As before, these orderings are purely local; they do not have to be consistent with one another. For the purposes of robustness bounds, two processors that share a resource are considered to be at distance one apart. This bipartite graph need not be physically present; it is enough that each processor have a local view of it.

We now define a particular labeling problem on the bipartite graph, called *happy coloring & marking*. Each processor is labeled with a *color* from $\{1, 2, \ldots, r\}$ and a *competitor set* C, and each edge is labeled with a *marking* from $\{0, 1\}$. Given such a labeling, we say that processors v and w are *competitors* if there is a resource ρ such that the edges (v, ρ) and (w, ρ) are both marked 1. A labeling is legal if:

1. each processor has at least m incident edges marked 1,
2. if processors v and w are competitors, then v and w are colored differently, and
3. the competitor set at v contains v's local names of the processors that are competitors with v.

The idea is that each v will compete for the resources connected to it by edges marked 1.

Assuming that $m \leq \lceil d/r \rceil$, an algorithm for happy coloring & marking is shown in Figure 3.

Theorem 5.1 *The algorithm in Figure 3 is a $(2, \nu)$-robust algorithm for happy coloring & marking.*

Proof. Assume that $B(v, 2)$ is stable for time ν. Note that a j in the computation of $color(v)$ exists because $m \leq \lceil d/r \rceil$. Two competitors cannot receive the same color j, because then they would both be the jth processor in the ordering at some resource. The computation of *color*(v) and the marking of edges incident on v requires no communication with other processors. In time ν, processors exchange marking information, so the competitor sets can be found. If $B(v, 2)$ remains stable, no neighbor of v changes the marking of its incident edges, so the label of v is stable. □

Given this coloring, we can now use the doorway algorithm of Figure 2. The only difference is that now the processors are r-colored, and the conflict graph is r-partite. An upper bound on the response time can be shown to be $O(rm^{r-1}(\tau + \nu))$. As in the proof of Theorem 4.1, let $T_{c,p}$ be the upper bound on the time for a processor with color c to complete phase p. Say that color i has priority over color j if $i < j$. By the same reasoning as in Theorem 4.1:

$$\begin{aligned} T_{1,2} &= \tau + 2\nu \\ T_{c,2} &= m(\max_{i<c} T_{i,2} + \tau + \nu) + 2\nu \quad \text{for } c > 1 \\ T_{c,1} &= \max_{i \neq c} T_{i,2} + \tau + 2\nu. \end{aligned}$$

This gives an upper bound $O(rm^{r-1}(\tau + \nu))$ on response time. The maximum length of a waiting chain is now $r + 1$.

Using Theorem 2.1, this gives the following.

Theorem 5.2 *For $m \leq \lceil d/r \rceil$, the composition of the algorithm in Figure 3 and the algorithm in Figure 2 is a $(r+3, O(rm^{r-1}(\tau+\nu)))$-robust algorithm for (d, m)-DPP in the case that each resource is shared by up to r processors.*

5.2 The impossibility result

Since we are proving impossibility in this section, there is no harm in assuming a strong computational model: The processors operate in lock-step synchrony with all messages taking one time unit to be delivered; the network and the connections of processors to resources does not change; each processor and resource has a unique numerical ID; and the eating time τ is one time unit.

As in [28], a *local algorithm with time bound t* for an LCL problem operates in parallel at every vertex v; the algorithm at vertex v collects information about the structure of the network and ID's in $B(v, t)$ and, based on that information, assigns an output label to v.

A useful tool for proving impossibility of local solution is *order invariance.* Intuitively, a local algorithm is *order-invariant* if the output label it produces does not depend on the actual value of the ID's in $B(v, t)$, but only on the relative ordering of these ID's. More formally, if η and η' are two ID labelings of the vertices of some $B(v, t)$, and if for every two vertices v and w, $\eta(v) < \eta(w)$ iff $\eta'(v) < \eta'(w)$, then the algorithm assigns the same label to v when given $B(v, t)$ labeled by either η or η'. By a Ramsey theory argument, building on similar applications of Ramsey theory as in, e.g., [15, 24, 33], Naor and Stockmeyer [28] show that if an LCL can be solved by a local algorithm in time t, then it can be solved by an order-invariant local algorithm in time t. This result is used in proving the impossibility result, stated next. The proof is similar to the proof for the case $r = 2$ in [28].

Theorem 5.3 *For $m > \lceil d/r \rceil$ and $\tau = \nu = 1$, there is no algorithm with constant response time for (d, m)-DPP if each resource is shared by r processors. This holds even in a static synchronous system with unique ID's on the processors and the resources.*

Proof. Fix some r and m. Since d is a lower bound on the number of resources connected to each processor, it suffices to do the proof with d as large as possible, that is, $d = r(m-1)$.

Imagine that the processors are placed at the vertices of a $k = (m-1)$-dimensional torus graph, and that the vertices are "named" by k-tuples, $(i_1, \ldots, i_k)$ where $0 \leq i_j < n$ for each j and for some large n. For each $(i_1, i_2, \ldots, i_k)$ and each j with $1 \leq j \leq k$, the r processors at the vertices $(i_1, \ldots, i_{j-1}, i, i_{j+1}, \ldots, i_k)$ for $i_j + 1 \leq i \leq i_j + r$ share a resource, where elements of a k-tuple are reduced modulo n. So each processor is connected to $rk = d$ resources, as required. We say that two processors are "neighbors" if the L_1 distance between their k-tuple names is 1.

We first show that a solution to the (d, m)-dining philosophers problem with constant response time would give a local solution to the following LCL problem, for some constant c depending only on m and r (a value for c is determined below):

1. Each vertex is labeled either 1 or 2;

2. Each k-dimensional $c \times c \times \cdots \times c$ submesh M contains some vertex labeled 1 and some vertex labeled 2.

Given an arbitrary ID-numbered torus, run the assumed (d, m)-dining philosophers algorithm starting in the configuration where all vertices are initially hungry. Recall that the vertices operate in lock-step synchrony and that each message delay is one time unit. When a vertex enters the eating state, it remains in the eating state for one time unit, and remains in the resting state thereafter. The following rules are used to determine the label of vertex v. Let s be the step at which v enters the eating state. If v has not received the message "eating" from one of its neighbors at step s or earlier, then v chooses the label 1, and sends the message "eating" to all of its neighbors at step s (in addition to any messages that the (d, m)-dining philosophers algorithm sends at this step); the "eating" message is received by v's neighbors at the next step $s+1$. Otherwise (v received an "eating" message at step s or earlier), then v chooses the label 2 and does not send an "eating" message. The labeling algorithm runs in constant time since the assumed (d, m)-DPP algorithm has constant response time. It remains to show that condition 2 above holds for a large enough c. Assume that $c \geq 3$. If there is a $c \times \cdots \times c$ submesh M with all vertices labeled 2, then there would be some v such that v and all of its neighbors are labeled 2. But this is impossible since, if all the neighbors of v are labeled 2, they do not send "eating" to v, so v will be labeled 1 at the step when it eats. If there is a $c \times \cdots \times c$ submesh M with all vertices labeled 1, then all vertices of M enter the eating state at the same step s. This gives a contradiction as follows. The number of resources connected to vertices of M is at most $kc^k + O(rkc^{k-1}) = (m-1)c^k + O(rkc^{k-1})$. But in order for each vertex of M to own at least m of these resources, there must be at least mc^k of them. We obtain a contradiction by choosing c large enough.

It is now easy to prove that this LCL cannot be solved locally. Suppose that it can be solved locally. By the order-invariance result mentioned above, there would be an order-invariant local algorithm A for the LCL, running in some constant time t. Assign unique

```
REPEAT FOREVER
    Find a 1-1 assignment $indexedge : towards(v) \mapsto \{1, \ldots, indegree(v)\}$;
    IF there exists a neighbor $w$ s.t. $color(v) \neq color(w)$ THEN
        $weakcolor(v) := color(v)$;
    ELSE
        $weakcolor(v) :=$ smallest $x \notin \{\, indexedge(e) \mid e \in from(v) \,\}$;
END
```

Figure 4: Weak coloring at vertex v

numerical ID's to the vertices according to the lexicographic order of their k-tuple names. Each resource can now be "named" by a unique set of r vertex ID's. Assign numerical ID's to the resources according to the lexicographic order of these sets. For n large enough, there will be some $c \times c \times \cdots \times c$ submesh M such that $B(v,t)$ looks the same to the order-invariant A for every vertex v in M. So A assigns the same output label to every vertex of M, which contradicts the definition of the LCL. $\Box$

6 Weak Coloring from Happy Coloring & Orientation

In this section we give another application of happy coloring & orientation, this time to the weak coloring problem of [28]. This problem is interesting since it was, to our knowledge, the first example of a nontrivial labeling problem that can be solved locally, i.e., by looking only at neighborhoods of constant radius. Using happy coloring & orientation has two advantages over the algorithm of [28]: first, the new algorithm is simpler; and second, the algorithm is robust in a dynamic network (whereas [28] considered only the static case). A weak c-coloring of a graph $G = (V, E)$ is an assignment $weakcolor : V \mapsto \{1 \ldots c\}$ such that for every vertex $v \in V$ there is a neighbor $w \in V$ such that $weakcolor(v) \neq weakcolor(w)$. Naor and Stockmeyer [28] have shown a local algorithm for finding a weak 2-coloring of graphs in which the degree of every vertex is odd. We now show that happy coloring & orientation is useful for finding a weak coloring. Given a happy coloring & orientation of a graph of maximum degree d, our algorithm finds a weak $(\lceil d/2 \rceil + 2)$-coloring in one step; v is legally weakly colored (has a neighbor colored differently) if v has odd degree. (It is known that there is no local solution to weak coloring if graphs can have vertices of even degree [28].) This in turn can be reduced to a weak 2-coloring in $\log^* d + O(1)$ additional steps, given the methods in [28].

We assume some happy coloring & orientation on G. Let $color(v)$ denote the color given to v by the happy coloring & orientation, and let $indegree(v)$ denote the indegree of v, $towards(v)$ denote the set of edges oriented towards v, and $from(v)$ denote the set of edges oriented from v in the happy coloring & orientation. The algorithm below assigns a function $weakcolor(v)$ to a vertex v. The algorithm also assigns $indexedge(e)$ for all the edges e incoming to v. A description of the algorithm can be found in Figure 4.

Theorem 6.1 *The composition of the algorithm in Figure 1 with the algorithm in Figure 4* *is a $(4, 3\nu)$-robust algorithm for a weak $(\lceil d/2 \rceil + 2)$-coloring on the vertices of odd degree.*

Proof. We show that the algorithm in Figure 4 is $(2, \nu)$-robust, given a happy coloring & orientation. The result then follows from Theorem 3.1 and Theorem 2.1.

Assume that $B(v, 2)$ remains stable for time ν with a correct happy coloring & orientation. Within time ν, all vertices in $B(v, 1)$ will have a common view of their colors and the value of $\mathit{indexedge}(e)$ for all edges incident on v. If v has a neighbor w s.t. $\mathit{color}(v) \neq \mathit{color}(w)$, then $\mathit{weakcolor}(v) \neq \mathit{weakcolor}(w)$, since one gets '$a$' and the other '$b$'. Suppose then that all of v's neighbors are colored with $\mathit{color}(v)$. If the degree of v is odd,

$$\mathit{indegree}(v) \geq \lceil d_v/2 \rceil > \mathit{outdegree}(v)$$

and hence there exists an $1 \leq x \leq \lceil d_v/2 \rceil$ such that x is not assigned as $\mathit{indexedge}(e)$ to any $e \in \mathit{from}(v)$. Consider the vertex w such that edge $(w, v) \in \mathit{towards}(v)$ and $\mathit{indexedge}((w, v)) = x$. It cannot be the case that $\mathit{weakcolor}(w) = x$, since x is *not* missing among the assignments to $\mathit{from}(w)$. Hence v is weakly colored properly.

As long as $B(2, v)$ remains stable, all neighbors w of v have stable $\mathit{color}(w)$ and no neighbor will change the value of $\mathit{indexedge}(e)$ for any $e \in \mathit{from}(v)$. $\square$

7 ID's vs. Orientation

All the algorithms presented so far relied on an initial (arbitrary) edge orientation and did not need the stronger assumption that each node has a unique ID. In this section we show an example where an initial orientation is not sufficient (even for static networks).

Consider the following variation on weak-coloring: All nodes v should receive a color in $\{1, \ldots, c\}$ such that if the number of nodes of distance *exactly* 2 from v is odd, then at least one of them is colored differently than v.

First note that with unique ID's assigned to every node the problem is solvable locally. All nodes simulate the local weak 2-coloring algorithm, substituting for edges paths of length exactly 2. The orientation of the virtual edge is determined by the ID's of its endpoints (or alternatively we can use the algorithm of [28]).

On the other hand, a labeling with the above properties cannot be found if the only means of breaking symmetry is an initial edge orientation (and port labeling). To see this, assume that there is a t-step algorithm for finding such a labeling, for some $t \geq 1$. We construct a graph G' where this algorithm necessarily fails. In describing G' we should provide the topology, the initial edge orientation and the port labeling at the nodes.

Start with the graph $G = (V, E)$ which is an unrooted tree of degree 3 at every internal node and has radius at least $t+2$. Label the ports at every node with $\{a, b, c\}$ in a consistent manner. I.e. if edge (u, v) is considered port a at u, then it is port a at v as well. Note that given such a labeling, all nodes of distance at least t from the leaves "see" the same t-neighborhood.

Transform G to $G' = (V', E')$ by the following rules: for every edge $(u, v) \in E$ generate two new nodes u' and v'; add nodes u, v, u' and v' to V' and the edges $(u, u'), (u, v'), (v, u')$ and (v, v') to E'. Orient the edges $u \mapsto u', u' \mapsto v, v \mapsto v', v' \mapsto u$. As for the port labeling, use labels from $\{a_1, a_2, b_1, b_2, c_1, c_2\}$. At nodes $u \in V$, each edge in E was replaced by one incoming edge and one outgoing edge. At these nodes, for the port labeled a in G label the port of the incoming edge with a_1 and the port of the outgoing edge with a_2. Similarly for the ports labeled b and c. As for the new nodes, label the port of the incoming edge with a_1 and that of the outgoing edge with a_2.

Note that for the nodes $u \in V$, the number of nodes of distance exactly 2 in G' is 3, i.e. the constraint on weak coloring applies to them. Also, it is still the case that in the graph G' all the nodes $u \in V$ of distance at least t from the leaves "see" the same t-neighborhood (with respect to orientation and port labeling). Thus there is no way to assign colors so that such a node has at least one node of distance exactly 2 which receives a different color.

We do not know what is the "weakest" assumption about symmetry that suffices for all local algorithms? Consider the following assumption: there is a hyper-edge orientation for all sufficiently large subsets. We suspect that all problems that can be solved locally assuming unique ID's can be solved under this assumption.

8 The Color Reduction Problem

In this section we investigate the following problem: starting with a legal coloring with c colors of the vertices of a graph (i.e. no two neighboring vertices receive the same color), what is the smallest number of colors c' such that every vertex can recolor itself legally with one of c' colors as a function of its *immediate* neighborhood only. The new coloring size is a function of c, the original number of colors, and d, the degree of the graph. I.e., the locality conditions we impose on the algorithm is more severe than in previous sections and we consider only single-step algorithms.

Cole and Vishkin [13] and Goldberg and Plotkin [16] suggested a method that yields a single-step color reduction from c colors to $(\log c)^d$ colors. Linial [20] showed a method, described below, that reduces c colors to $d^2 \log c$ colors. Szegedy and Vishwanathan [32] show non-constructively that the number of colors can be reduced to $O(d2^d \log\log c)$ colors. We show how to make their method constructive. Our method follows theirs closely and is made constructive by the application of (n,d)-universal sets or d-intersection independent sets (see definition below). The constructive solution has the same double-logarithmic dependence on c, although a slightly worse dependence on d, using the currently best known constructions of d-intersection independent sets.

8.1 Set systems and recoloring

Linial [20] suggested a recoloring method based on intersection properties of subsets. To recolor a graph, originally colored with c colors and with max degree d, we need a collection of c subsets $S_1, S_2, \ldots, S_c$ of some ground set B of size k. The desired property is that for any $i_0, i_1, \ldots, i_d \in \{1 \ldots c\}$, such that $i_0 \neq i_j$ for $1 \leq j \leq d$, we have that

$$S_{i_0} \not\subset \bigcup_{j=1}^{d} S_{i_j} \tag{1}$$

Such a set system allows us to recolor with $k = |B|$ colors: the recoloring at vertex v, initially colored i_0 and with neighbors colored $i_1, \ldots, i_d$, is done by choosing an $x \in S_{i_0}$

not in $\bigcup_{j=1}^{d} S_{i_j}$. From the property (1) we are guaranteed that such an x exists. Any two neighbors v and w colored with i_v and i_w choose distinct colors: v selects from a subset of $S_{i_v} - S_{i_w}$ and w selects from a subset of $S_{i_w} - S_{i_v}$.

Since the number of new colors is the size of the ground set B, we would like to find a construction with B as small as possible. From [14] it follows that the smallest such k is of size $d^2 \log c$. Szegedy and Vishwanathan [32] suggested that a different property of set systems is relevant to the color reduction problem and can yield a more efficient (for some values of c and d) recoloring. What we show below is that this property of set systems can be replaced by d-intersection independence. Rather than describe the method of [32] and how it is related to d-intersection independence, it is simpler to describe the color reduction method directly in terms of d-intersection independence.

8.2 Intersection independence

For a ground set B and subset $S \subset B$ let S^0 denote the complement of S, i.e. $B - S$, and let S^1 denote S itself. A collection of sets $S_1, S_2, \ldots, S_m \subset B$ is *d-intersection independent* if: For all $1 \le i_1 < i_2 < \cdots < i_d \le m$ and $b_1, b_2, \ldots, b_d \in \{0,1\}$

$$\bigcap_{j=1}^{d} S_{i_j}^{b_j} \neq \phi$$

Kleitman and Spencer [17] showed that the smallest k, for which there exists a set system of m subsets of a ground set B of size k which is d-intersection independent, is $2^{\theta(d)} \log m$. The upper bound is based on a probabilistic construction which, as noted in [25], can be made constructive using small-bias probability spaces which we now describe.

Small bias probability spaces: Let Ω be a probability space with m random variables $X_1, X_2, \ldots, X_m$. Ω is a *d-wise ϵ-bias probability space* if for any subset S of random variables from $X_1, X_2, \ldots, X_m$ of size at most d we have

$$|Prob[\bigoplus_{i \in S} X_i = 0] - Prob[\bigoplus_{i \in S} X_i = 1]| \le \epsilon.$$

An important property of d-wise ϵ-bias probability spaces is: for any subset S of random variables from $X_1, X_2, \ldots, X_m$ of size $1 \le \ell \le d$, the probability that the random variables of S attain a certain configuration deviates from $1/2^\ell$ by at most ϵ. Therefore d-wise ϵ-bias probability spaces are described as *almost d-wise independent.*

The size of the constructions of small bias probability spaces is $Poly(d, 1/\epsilon, \log m)$. More precisely, there are constructions of size:

- $O(\frac{d \log m}{\epsilon^3})$ – Naor & Naor [25] (optimized in Alon et al. [4])
- $O(\frac{d^2 \log^2 m}{\epsilon^2})$ – Alon et al. [5]

If $\frac{1}{\epsilon} = 2^d + 1$ then the size of the first probability space is $2^{O(d)} \log m$.

In order to obtain the d-intersection independent collection, set $\epsilon < 2^{-d}$ (say $\frac{1}{\epsilon} = 2^d + 1$). From a d-wise ϵ-bias probability space with m random variables we can get a d-intersection independent collection:

- k = size of probability space
- points of probability space = elements of ground set $B = \{1, \ldots, k\}$
- $S_i = \{$all points such that $X_i = 1\}$

Recently Naor et al. [27] showed a better construction for d-intersection independent collections which is of size $d^{O(\log d)} 2^d \log m$. The disadvantage of their construction is that it is a global one, in the sense that all the sets must be constructed together by a sequential algorithm of complexity similar to the size of the collection. However, this seems to be good enough for the purpose of designing local algorithms, since it can be thought of as a "compilation phase".

8.3 Efficient recoloring

Let $m = \lceil \log c \rceil$ and construct m subsets that are d-intersection independent. Let $c(v)$ denote the color of vertex v and for $1 \leq j \leq d$ let $c_j(v)$ denote the color of the jth neighbor of v, where a color is a string of m bits. A vertex v with $c(v) = b_1 b_2 \ldots b_m$ is recolored according to the following:

1. Find $i_1, i_2, \ldots, i_d$ where i_j is the index of first bit where $c(v)$ and $c_j(v)$ differ.
2. recolor v with $x \in \bigcap_{j=1}^{d} S_{i_j}^{b_{i_j}}$.

The number of colors used following the recoloring is therefore $k = O(d 2^{3d} \log m) = O(d 2^{3d} \log \log c)$ (or $d^{O(\log d)} 2^d \log \log c$ in case we apply the construction of [27]) and the claim below establishes that it is a proper recoloring procedure.

Claim 8.1 *The above scheme is a correct recoloring.*

Proof. First note that though $i_1, i_2, \ldots, i_d$ of step 1 need not be distinct, in the intersection $\bigcap_{j=1}^{d} S_{i_j}^{b_{i_j}}$ of step 2 there do not appear two sets S_l^0 and S_l^1 for $1 \leq l \leq m$: if $i_j = i_l$ then $S_{i_j}^{b_{i_j}} = S_{i_l}^{b_{i_l}}$, since $b_{i_j} = b_{i_l}$. Therefore, from the d intersection property there is always an $x \in \bigcap_{j=1}^{d} S_{i_j}^{b_{i_j}}$ and hence step 2 can be executed. As for correctness, we should show that for all $1 \leq j \leq d$, if $c(v) \neq c_j(v)$ then v and its jth neighbor are recolored with different colors. We claim that v and its jth neighbor choose new colors from disjoint subsets of B: v will choose from a subset of $S_{i_j}^{b_{i_j}}$ and the jth neighbor of v will chose from a subset of $S_{i_j}^{1-b_{i_j}}$. $\square$

Example. We show a construction for $c = 8$ and $d = 2$. The algorithm reduces the number of colors from 8 to 4. Let $m = 3$, $k = 4$ and $B = \{1, 2, 3, 4\}$. The system $S_1 = \{3, 4\}$, $S_2 = \{2, 3\}$ and $S_3 = \{2, 4\}$ is 2-intersection independent. For a vertex colored 011 with neighbors colored 101 and 010 the algorithm assigns $i_1 = 1$ and $i_2 = 3$. Therefore it looks for a new color in $S_1^0 \cap S_3 = \{2\}$. The new color is therefore 2.

3-coloring a ring: Suppose now that the above method for color reduction is applied successively on the ring, i.e. a graph of degree 2, in order to obtain a legal 3-coloring. I.e., we start with a coloring of c colors and in each step replace the color of each vertex according to the above algorithm. Note that this method reduces the number of colors for any $c > 4$ (see the example). When the number of colors is 4, it can be reduced to 3 by having each vertex colored 4 choose a color in $\{1, 2, 3\}$ not used by its neighbors (as in [16]). Since each round reduces the number of colors used to order of $\log \log$ the number of colors in the previous round, the number of steps this procedure takes till a 3-coloring is found is in $(\frac{1}{2} + o(1)) \log^* c$. This bound is tight up to the $o(1)$ term [20] (see also [26]).

Acknowledgments.

We thank the anonymous referees for their useful comments.

References

[1] Y. Afek, B. Awerbuch, and E. Gafni, Applying static network protocols to dynamic networks, *Proc. 28th IEEE Symposium on Foundations of Computer Science*, 1987, pp. 358–370.

[2] Y. Afek and E. Gafni, End-to-end communication in unreliable networks, *Proc. 2nd ACM Symposium on Principles of Distributed Computing*, 1988, pp. 131-148.

[3] Y. Afek, S. Kutten, and M. Yung, Memory-efficient self-stabilizing protocols for general networks, *Proc. 4th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science*, Vol 486, Springer-Verlag, New York, 1990, pp. 15–28.

[4] N. Alon, J. Bruck, J. Naor, M. Naor, and R. Roth, Construction of asymptotically good, low-rate error-correcting codes through pseudo-random graphs, *IEEE Trans. Information Theory* 38 (1992), pp. 509-516.

[5] N. Alon, O. Goldreich, J. Hastad, and R. Peralta, Simple constructions of almost k-wise independent random variables, *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990, pp. 544–553. See also: *Random Structures and Algorithms* 3 (1992), pp. 289–304.

[6] B. Awerbuch, L. Cowen, and M. Smith, Efficient asynchronous distributed symmetry breaking, *Proc. 26th ACM Symposium on Theory of Computing*, 1994, pp. 214–223.

[7] B. Awerbuch and S. Even, Reliable broadcast protocols in unreliable networks, *Networks* 16 (1986), pp. 381-396.

[8] B. Awerbuch, O. Goldreich, and A. Herzberg, A quantitative approach to dynamic networks, *Proc. 9th ACM Symposium on Principles of Distributed Computing*, 1990, pp. 189–203.

[9] B. Awerbuch, B. Patt-Shamir, and G. Varghese, Self-stabilization by local checking and correction, *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, 1991, pp. 268–277.

[10] J. Bar-Ilan and D. Peleg, Distributed resource allocation algorithms, *Proc. 6th International Workshop on Distributed Algorithms*, 1992, pp. 277–291.

[11] K.M. Chandy and J. Misra, The drinking philosophers problem, *ACM Trans. Programming Languages and Systems* 6 (1984), pp. 632–646.

[12] M. Choy and A.K. Singh, Efficient fault tolerant algorithms for resource allocation in distributed systems, *Proc. 24th ACM Symposium on Theory of Computing*, 1992, pp. 593–602.

[13] R. Cole and U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, *Information and Control* 70 (1986), pp. 32–53.

[14] P. Erdös, P. Frankl, and Z. Füredi, Families of finite sets in which no set is covered by the union of r others, *Israel J. of Math.* 51 (1985), pp. 79–89.

[15] G.N. Frederickson and N.A. Lynch, Electing a leader in a synchronous ring, *J. Assoc. Comput. Mach.* 34 (1987), pp. 98–115.

[16] A.V. Goldberg and S.A. Plotkin, Parallel $(\Delta + 1)$-coloring of constant-degree graphs, *Inform. Proc. Lett.* 25 (1987), pp. 241–245.

[17] D.J. Kleitman and J. Spencer, Families of k-independent sets, *Discrete Math.* 6 (1973), pp. 255–262.

[18] S. Kutten and D. Peleg, Fault-local distributed mending, *Proc. 14th ACM Symposium on Principles of Distributed Computing*, 1995, pp. 20–27.

[19] S. Kutten and D. Peleg, Tight fault locality, *Proc. 36th IEEE Symposium on Foundations of Computer Science*, 1995, pp. 704–713.

[20] N. Linial, Locality in distributed graph algorithms, *SIAM J. Comput.* 21 (1992), pp. 193–201.

[21] N. Linial, Local-global phenomena in graphs, Technical Report 93-9, Hebrew University, Institute of Computer Science, 1993.

[22] N. Linial, D. Peleg, Yu. Rabinovich, and M. Saks, Sphere packing and local majorities in graphs, *Proc. 2nd Israeli Symp. on Theory of Computing and Systems*, 1993, pp. 141–149.

[23] N.A. Lynch, Upper bounds for static resource allocation in a distributed system, *J. Comput. System Sci.* 23 (1981), pp. 254–278.

[24] S. Moran, M. Snir, and U. Manber, Applications of Ramsey's theorem to decision tree complexity, *J. Assoc. Comput. Mach.* 32 (1985), pp. 938–949.

[25] J. Naor and M. Naor, Small-bias probability spaces: efficient constructions and applications, *SIAM J. Comput.* 22 (1993), pp. 838–856.

[26] M. Naor, A lower bound on probabilistic algorithms for distributive ring coloring, *SIAM J. Disc. Math.* 4 (1991), pp. 409–412.

[27] M. Naor, L. J. Schulman, and A. Srinivasan, Splitters and near-optimal derandomization, *Proc. 36th IEEE Symposium on Foundations of Computer Science*, 1995, pp. 182–191.

[28] M. Naor and L. Stockmeyer, What can be computed locally?, *SIAM J. Comput.* 24 (1995), pp. 1259–1277.

[29] A.A. Schäffer and M. Yannakakis, Simple local search problems that are hard to solve, *SIAM J. Comput.* 20 (1991), pp. 56–87.

[30] M. Schneider, Self-stabilization, *ACM Computing Surveys* 25(1) (1993), pp. 45–67.

[31] E. Styer and G.L. Peterson, Improved algorithms for distributed resource allocation, *Proc. 7th ACM Symposium on Principles of Distributed Computing*, 1988, pp. 105–116.

[32] M. Szegedy and S. Vishwanathan, Locality based graph coloring, *Proc. 25th ACM Symposium on Theory of Computing*, 1993, pp. 201–207.

[33] A.C.-C. Yao, Should tables be sorted?, *J. Assoc. Comput. Mach.* 28 (1981), pp. 615–628.