On the distributed complexity of computing maximal matchings

Michał Hańćkowiak
Dept of Math and CS
Adam Mickiewicz University
Poznań, Poland

Michał Karoński
Dept of Math and CS
Adam Mickiewicz University
Poznań, Poland
&
Dept of Math and CS
Emory University
Atlanta, Georgia, USA

Alessandro Panconesi
DSI, *La Sapienza*
via Salaria 113, 00198 Roma, Italy

Abstract

We show that maximal matchings can be computed deterministically in $O(\log^4 n)$ rounds in the synchronous, message-passing model of computation. This is one of the very few cases known of a non-trivial graph structure, and the only "classical" one, which can be computed distributively in polylogarithmic time without recourse to randomization.

1 Introduction

One of the fascinating questions of computer science is whether and to what extent randomization increases the power of algorithmic procedures. It is well-known that, in general, randomization makes distributed algorithms more powerful, for there are examples of basic coordination tasks in asynchronous systems that cannot be solved by deterministic procedures but admit simple randomized solutions. Randomization is also demonstrably more powerful in synchronous systems, as shown by the important example of oblivious routing in the hypercube (see, for instance, [15, 21]). In this paper we are interested in this question in the context of distributed graph algorithms, where a synchronous, message-passing network *without shared memory* is to compute a function of its own topology, and focus on the problem of computing maximal matchings. We show that maximal matchings can be computed in polylogarithmically-many communication rounds by *deterministic* distributed algorithms. So, as far as maximal matchings are concerned, randomization is not necessary to go over the sub-linear "divide".

To put our work into perspective we review some of the relevant facts and literature.

In a distributed network or architecture without shared memory the cost of sending a message between two nodes is proportional to their distance in the network. Since sending messages to far-away nodes is expensive, it is desirable that computation be based only on information available locally. This locality constraint can be quite severe when one is to compute a global function of input data that are spread across the network, and represents a challenge from the point of view of algorithmic design. This communication problem is completely neglected in the popular PRAM model. There, the existence of a shared memory which can be accessed in unit time allows fast collection and dissemination of data among the processors. Once this assumption is removed and the cost of communication is taken into consideration, several computational problems which were easily solvable suddenly become hard or unsolvable efficiently, especially if one is seeking *deterministic* solutions.

The study of distributed graph algorithms goes back to (at least) the work of Linial [17] where an $\Omega(\log^* n)$ lower bound for computing maximal independent sets (MIS's) in the ring is given. Together with the $O(\log^* n)$ upper bound given by a beautiful algorithm of Cole and Vishkin, this is one of the all too rare examples in complexity theory where the complexity of a computational problem can be determined exactly (modulo constants). Interestingly, it can be shown that randomization does not help [22].

Generalizing from rings to bounded degree graphs one sees that several classical graph structures of both theoretical and practical interest, including maximal independent sets (MIS's), maximal matchings, $(\Delta+1)$- and even Δ-vertex colourings, can be computed in polylogarithmic time [1, 2, 7, 26] (Δ denotes the maximum degree of the input network). In fact, many of these algorithms are very satisfactory because they are both quite simple and really of low complexity, i.e. with small exponents and no hidden large constants.

Further generalizing from bounded degree graphs to general topologies has proven elusive, in spite of several efforts [1, 2, 18, 23, 26, 27]. The situation here is, more or less, as follows. For a reasonably large class of graph structures, the asymptotically best *deterministic* algorithm known to date uses $O(n^{\epsilon(n)})$ rounds, where $\epsilon(n)$ is a function which (very slowly) goes to 0 as n, the size of the network, grows. These solutions are mainly of theoretical interest, since the protocols are quite cumbersome and their implementation would probably be prohibitively expensive. On the other hand, once randomization is allowed, the same graph structures can be computed in polylogarithmically-many rounds. Furthermore, these randomized algorithms are usually extremely simple and their actual complexity is very low. For instance, $(\Delta+1)$–vertex colouring and MIS can be computed in $O(\log n)$ rounds with high probability by exceedingly simple protocols [19, 20, 24].

Another important example of something that can be computed in polylogarithmic time by using randomness by distributed algorithms, while it is not known whether the same time bounds can be attained without it, is that of $(O(\log n), O(\log n))$–decompositions an interesting type of graph decomposition with several applications [1]. Using randomization, these structures can be computed in $O(\log^2 n)$ rounds [18]. In fact, there exist non-trivial functions, such as nearly optimal edge colourings, that can be computed, with high probability, by extremely simple, indeed trivial, randomized algorithms in $o(\log n)$ (little-oh of n) rounds or even, under suitable degree assumptions, in as few as $O(\log\log n)$ rounds [8].

The question then is whether, in the context of distributed graph algorithms, randomization is necessary in order to obtain protocols that run in polylogarithmically-many rounds in the size of the network.

In an attempt to gain some insight into this problem, we show that for a non-trivial and important graph structure, maximal matchings, randomization is not needed. Matchings are important structures from a theoretical point of view but might also be of practical interest, since, in some situations, they correspond to a set of operations, say, data transfers, that can be performed simultaneously without mutual interferences. We note that maximal matching is a special case of the difficult open problem of determining whether MIS's can be quickly computed deterministically in spite of the locality constraint. The solution presented here takes $O(\log^4 n)$ many rounds, improving an earlier result by the same authors [9]. Although the time complexity of our protocol is quite high, it should be remembered that even in the EREW-PRAM model, where information can be distributed inexpensively via the shared memory, the best asymptotic complexity for computing maximal matchings is $O(\log^3 n)$ [11]. Our solution hinges on a distributed procedure for bipartite graphs which, for almost all vertices in the graph, cuts the degree of a vertex almost perfectly in half. This approximate degree splitter might be useful in other contexts.

To our knowledge, maximal matchings are one of the very few examples of non-trivial graph functions which can be computed deterministically in polylogarithmically-many communication rounds in the distributed model, without additional assumptions on the input network. Other notable exceptions are the so-called ruling forests of [1] and the k-dominating sets of [14] both of which, however, are not "classical" graph structures.

We end this section by spelling out our model of computation, the *synchronous, message-passing distributed network*. Here, a distributed network (or architecture) is modeled as an undirected graph. The vertices of the graph correspond to processors and edges correspond to bi-directional communication links. The network is synchronous in the sense that computation takes place in a sequence of *rounds*; in each round, each processor reads messages sent to it by its neighbours in the graph, does any amount of local computation, and sends messages back to each of its neighbours. The time complexity of a distributed algorithm is then given by the number of rounds needed to compute the desired function. Each node of the network has a unique identifier (ID) and knows it. We assume that the ID's of the network are the integers from 1 to n.

The problem we study is this: A distributed network is to compute a maximal matching of its own (unknown) topology.

2 Preliminaries

We shall make use of standard graph theoretic terminology. In particular, given a set of edges X in a graph G, $G[X]$ denotes the subgraph of G induced by X with vertex set $V(G)$.

Definition 1 *The* weight *of an edge uv in a graph G, denoted as $w_G(uv)$, is equal to the number of edges incident to it.*

Definition 2 *Given a graph G and a set of edges $X \subseteq E(G)$, $touch(X)$ denotes the set of edges which are "touched" by X, i.e., the set X itself union the set of edges incident to some edge of X.*

A *degree-2 graph* is a graph G of maximum degree 2, i.e., a collection of paths and cycles. Let G be a degree-2 graph and suppose that it is equipped with a 3-vertex colouring. Then, a maximal matching M of G can be computed in constant many rounds, as follows. Each vertex of colour 1 selects one of its incident edges arbitrarily; let A be the set of edges so selected. The set A induces a graph consisting of isolated edges and paths of length two. The middle point of each such path selects one of its two incident A-edges arbitrarily. This leaves a set $A' \subseteq A$ which is a matching with respect to G and a maximal matching with respect to (the graph induced by) A. The edges in $touch(A')$ are removed from G, leaving a leftover graph G'. Now, the same procedure is repeated with the vertices of colour 2. They select a set B of edges and from it, a matching $B' \subseteq B$ is computed as before. Note that B' is disjoint from A'. Again, $touch(B')$ is removed from G', leaving a graph G''. This graph is itself a collection of isolated edges and paths of length two because all vertices that have not lost any of its two edges have color 3 and spaces between them are greater than 3. Therefore a maximal matching C' (with respect to G'') can again be computed as before. Since $touch(C') \supseteq E(G'')$ there are no more edges left. The set $M := A' \cup B' \cup C'$ is therefore, by construction, a maximal matching with respect to G.

In [7] and [6] two deterministic, distributed algorithms are given to compute a $(\Delta + 1)$-vertex colouring of an input graph of maximum degree Δ, where Δ is constant (the algorithm in [7] does not need this assumption). Their time complexity is, respectively, $O(\Delta \log n)$ and $O(\Delta^{\Delta} \log^* n)$ many communication rounds, where n is the number of vertices of the input graph. Both algorithms are very simple in the sense that they can be easily implemented in our distributed model.

Altogether, this means that maximal matchings of degree-2 graphs can be computed in $O(\log n)$, or even $O(\log^* n)$, many communication rounds (in fact, the result holds for any constant degree). We record this fact for future reference.

Fact 3 *Maximal matchings of degree-2 graphs can be computed in $O(\log n)$ many communication rounds in the synchronous, message-passing model of computation.*

3 Reducing the Problem to Two-coloured Bipartite Graphs

Suppose that we have a distributed procedure that, given a two-coloured, bipartite graph, computes a matching that matches a constant fraction of the edges, in $O(t(n))$ many rounds. Here and in the rest of the paper, "two-coloured" means that the vertices know which side of the bipartition they belong to. In this section we show that, assuming this, there is a distributed procedure that, given any input graph, computes a maximal matching in $O(\log n\ t(n))$ many rounds.

We start with an outline of the algorithm which is meant to highlight the intuition behind our approach and, hopefully, to pave the way for the precise analysis to follow. The basic structure of the algorithm is standard. The maximal matching is computed incrementally, in *phases*. The goal of a phase is to compute a matching $\mathcal{M}$ such that $|touch(\mathcal{M})|$ is a constant fraction of $|E(G)|$. The edges of $\mathcal{M}$ are added to the partial solution computed so far, and are removed together with the edges incident to them. The left-over graph is the input to the next phase. This takes $O(\log n)$ phases since, by assumption, $|touch(\mathcal{M})|$ is a constant fraction of the number of edges of the input graph. The heart of the algorithm is a phase, which is as follows. A phase begins by computing an arbitrary orientation of the edges of the input graph G, after that every vertex splits itself into two vertices called *siblings*. One sibling inherits all of the incoming edges, while the other inherits all of the outgoing edges. We therefore obtain a bipartite graph $\mathcal{B}$ whose sides of the bipartition are the "in" siblings and the "out" siblings. Notice that G and $\mathcal{B}$ have the "same" edge set. In the bipartite graph $\mathcal{B}$ the orientation of the edges is ignored (or, if you prefer, removed) and, resorting to the above mentioned procedure, a matching $\mathcal{L}$ is computed that matches a constant fraction of the edges of $\mathcal{B}$. Now, since the edge sets of the bipartite graph and the input graph coincide, in actuality $\mathcal{L}$ matches a constant fraction of the edges of the entire input graph. The problem is that $\mathcal{L}$ is a matching in $\mathcal{B}$ but not in G! The next step is to select a "real" matching $\mathcal{M} \subseteq \mathcal{L}$ such that $|touch(\mathcal{M})|$ is a constant fraction of $|touch(\mathcal{L})|$, i.e. $\mathcal{M}$ still matches a constant fraction of the edges of G, but it is a true matching. This is achieved in the following way. A new, degree-2 graph C is generated from $\mathcal{L}$ by "merging" each pair of siblings back together. C is a collection of paths and cycles and its edge set is exactly $\mathcal{L}$. A matching $\mathcal{M}$ is then computed inside C with the property that $|touch(\mathcal{M})|$ is a constant fraction of $|touch(E(C))|$ and therefore a constant fraction of $|E(G)|$ since $E(C)$ and $\mathcal{L}$ are, essentially, the same set. To compute $\mathcal{M}$, a maximal matching J (as in junk) is computed in C. This can be done in $O(\log n)$ time, by Fact 3. The edges in J are discarded, leaving a collection of isolated edges and paths of length two. From this, a matching $\mathcal{M}$ is computed by including into it all isolated edges and, for each path of length two, the edge that, in G, has the highest weight. This ends the phase.

A counting argument shows that $\mathcal{M}$ matches a constant fraction of $|touch(\mathcal{L})|$, thereby matching a smaller, but still constant, fraction of $|E(G)|$. Therefore, $O(\log n)$ many phases suffice to compute a maximal matching in the entire graph.

After this outline we present the algorithm more precisely and argue its correctness.

Procedure Match

$\mathcal{F} := \emptyset$ // $\mathcal{F}$ will be the final matching

repeat

1. Direct every edge arbitrarily.
2. Each vertex u splits into two siblings u_{in} and u_{out}. All incoming edges of u become (non-oriented) edges incident to u_{in}, while all outgoing edges of u become (non-oriented) edges incident to u_{out}. The resulting bipartite graph is denoted by $\mathcal{B}$.
3. Invoke procedure BIPARTITEMATCH, described in Section 4, to compute a matching $\mathcal{L}$ inside $\mathcal{B}$.
4. Compute a degree-2 graph C as follows: Each pair of siblings u_{in} and u_{out} is merged back into u. Edges of $\mathcal{L}$ that were incident to u_{in} or u_{out} become edges incident to u.
5. Let D be the graph obtained from C by removing the vertices of degree one. Compute a maximal matching J of D. Consider the set of edges $E(C) - J$. This set consists of isolated edges and paths of length two. Each middle point of every such length-2 path removes the edge incident on itself that has the smallest weight in the current graph G_i. Let $\mathcal{M}$ be the set of remaining edges.
6. Add $\mathcal{M}$ to $\mathcal{F}$, the partial solution computed so far, remove $\mathcal{M}$ and its neighbouring edges, and repeat with the left-over graph as the input to the next phase, i.e.,

$$G_{i+1} := G_i[E(G_i) - \mathit{touch}(\mathcal{M})].$$

until the graph has no more edges.

In procedure MATCH each vertex keeps executing the algorithm has long as its degree in the current graph G_i is at least one. Assuming that procedure BIPARTITEMATCH can be implemented in the distributed model, it is apparent that procedure MATCH is a distributed algorithm. We shall prove that the number of iterations of the repeat-until loop is $O(\log n)$. We remark that the vertices must know the value of n. The exact number of phases can be divined from Lemma 4 below and from Lemma 11 and 12, and is equal to $299 \log_2 n$ because $\mathit{touch}(\mathcal{M}) \geq \frac{1}{216}|E(G)|$ in each phase of the procedure MATCH where G is the input graph to that phase. In the sequel, with slight abuse of notation we will identify $E(G)$ with $E(\mathcal{B})$ and $\mathcal{L}$ with $E(C)$. Let us now argue the correctness of the algorithm. By construction, it is apparent that $\mathcal{M}$ is a matching. The maximality of $\mathcal{F}$ follows from the fact that, at the end of each phase, only the edges in $\mathit{touch}(\mathcal{M})$ are removed.

We now turn to the analysis of the running time. Since, by assumption, the matching $\mathcal{L}$ returned by BIPARTITEMATCH "touches" a constant fraction of $|E(G)|$, the crux of the matter is to show that $|\mathit{touch}(\mathcal{M})|$ is a constant fraction of $|\mathit{touch}(\mathcal{L})|$. This would ensure that within $O(\log n)$ phases $\mathit{touch}(\mathcal{F}) = E(G)$, i.e., $\mathcal{F}$ is maximal. We do this next.

Lemma 4 *Let G be the graph input to the current phase of procedure* MATCH, *and let $\mathcal{M}$ be the matching computed at the end of the phase. Assume that procedure* BIPARTITEMATCH *computes a matching $\mathcal{L}$ of the bipartite graph $\mathcal{B}$ such that $|\mathit{touch}(\mathcal{L})|$ is constant fraction of $|E(\mathcal{B})|$. Then, $\mathcal{M}$ matches a constant fraction of the edges of G.*

Proof Recall that $E(C) = \mathcal{L}$ and that J is a matching. Since, in Step 5, the graph D is obtained from C by removing vertices of degree one,

$$\mathit{touch}(\mathcal{L}) = \mathit{touch}(E(C)) = \mathit{touch}(E(C) - J)$$

Then, since in Step 5 each middle point selects the edge of largest weight,

$$|touch(\mathcal{M})| \geq \frac{|touch(E(C) - J)|}{2} = \frac{|touch(\mathcal{L})|}{2}.$$

The claim follows from the assumption that $|touch(\mathcal{L})|$ is a constant fraction of $|E(\mathcal{B})|$ and the fact that $E(G) = E(\mathcal{B})$. $\bigtriangledown$

The time complexity of a single phase of procedure MATCH can be estimated as follows. Step 3 takes $O(t(n)) = O(\log^3 n)$ many rounds, by hypothesis. Step 5 takes $O(\log n)$ many rounds, by Fact 3. All remaining steps take constant many rounds. Overall we get

$$T(\text{MATCH}, n) = O(\log n \times t(n)) = O(\log^4 n)$$

many rounds for the whole of procedure MATCH. In the next section we shall prove that the number of rounds for procedure BIPARTITEMATCH is indeed $O(\log^3 n)$, thereby proving the following theorem.

Theorem 5 *On any given input graph G, procedure* MATCH *computes a maximal matching of G in $O(\log^4 n)$ many communication rounds in the synchronous, distributed model of computation. For the algorithm to work, each vertex must know the value of $n := |V(G)|$.*

4 Computing MM's in Two-coloured Bipartite Graphs

In this section we show how to compute a maximal matching of a given bipartite graph $G(L, R, E)$ where, importantly, each vertex knows which side of the bipartition it belongs to (L stands for left and R for right). That is, adopting our terminology, G is two-coloured. This assumption is verified by the bipartite graph $\mathcal{B}$ generated by procedure MATCH. The basic idea of the algorithm is to partition the edges of the graph into subgraphs called blocks, to be defined more precisely later, and to compute "good" matchings inside each block, in parallel. The matchings computed in each block are then put together to form a new graph. Since the blocks are edge disjoint but not vertex disjoint, the new graph is not itself a matching. In order to select a "sizable" matching inside it, we adopt a greedy strategy. Roughly speaking, the blocks are ranked at the outset in such a way that an edge coming from a "high order" block will match at least half as many edges as any subset of edges coming from any collection of "low-order" blocks. We now describe the solution in detail, starting with some definitions. Keep in mind that we are working inside a two-coloured, bipartite graph $G = (L, R, E)$.

Definition 6 $H_i = \{u \in L : \frac{n}{2^{i+1}} < d(u) \leq \frac{n}{2^i}\}$.

Remark: If we replaced n with $\Delta(G)$ in this definition, the resulting algorithm would have an improved running time (a $\log n$ factor replaced by a $\log \Delta$ factor), but this would require the additional knowledge of Δ.

Notice that the H_i's, for $0 \leq i \leq \lfloor \log_2 n \rfloor$, partition the set L of left-hand-side vertices.

Definition 7 *The* block *induced by H_i, denoted as B_i, is the subgraph of G induced by the edges incident to H_i. The vertex set of B_i is $H_i \cup N(H_i)$, where the latter is the set of neighbours of H_i. Obviously, $H_i \subseteq L$ and $N(H_i) \subseteq R$.*

The blocks B_i's, for $0 \le i \le \lfloor \log_2 n \rfloor$, partition both $E(G)$ and the set of vertices L. Notice that a partition of G into blocks can be generated in constant time, provided that the value of n (or, alternatively that of $\Delta(G)$) is known. Two different blocks, say, B_i and B_j, can have right-hand-side vertices in common. In the algorithm to follow, a matching is computed simultaneously in all blocks. If a vertex of R belongs to two or more blocks, it will virtually "duplicate" itself and participate in the computations of the matchings in all blocks to which it belongs.

To compute a matching in a block, edges are removed in phases, until we are left with a matching. The matching has the property that if it is removed together with the edges incident to it, a constant fraction of the edges in the block will disappear. After these matchings are computed in parallel in all blocks, a final matching is computed that matches a constant fraction of the edges in the whole graph. Since a vertex $v \in R$ can participate to more blocks simultaneously, it is possible that at the end there is more than one edge incident to it, coming from matchings of different blocks. To select a final matching for the whole graph an edge is selected greedily– the edge selected is the one whose insertion into the final matching would remove the maximum number of edges; it is the edge with the maximum number of incident edges or, in other words, the edge uv with the maximum value of $d(u)$. This simple locally greedy approach works– i.e., it results in a matching incident to a constant fraction of edges overall– because of the way the sets H_i are defined. Namely, consider a sequence $i < j_1 < j_2 < \ldots < j_k$ then, the smallest degree of a vertex in H_i is at least half the sum of the degrees of vertices each of which comes from H_{j_s}, $1 \le s \le k$. We now proceed to explain how a good matching inside a given block can be computed.

Given a subgraph $X \subseteq G(L, R, E)$, we shall use the shorthand notation $l(X) := V(X) \cap L$ and $r(X) := V(X) \cap R$. The degree of a vertex u inside X is denoted by $d_X(u)$. The next definition is pivotal.

Definition 8 *An* (a,d)-spanner *of a block* $B = (H_i, N(H_i), E_B)$ *is a subgraph* $S \subseteq B$ *such that:*

- $|l(S)| \ge a|H_i|$, *i.e.* S *contains a constant fraction of the vertices of* H_i.
- *For every* $u \in l(S)$, $d_S(u) \in [1,d]$, *i.e. the degree of left-hand-side vertices is constant; and,*
- *For every vertex* $v \in r(S)$, $d_S(v) \le 2^{-k(B)} d_B(v) + 1$, *where* $k(B) := \lceil \log_2 n \rceil - i - 4$.

A spanner is a subgraph of "small" degree that spans a "large" fraction of vertices in H_i. As we shall see, given a spanner in a block it is easy and inexpensive to compute a matching inside the spanner that "touches" a constant fraction of the edges of the block.

Roughly speaking, a spanner in a block is computed by slashing repeatedly the vertex degrees. In the process we want to ensure that "almost all" of the vertices of H_i have their degree "almost perfectly" cut in half at each iteration of the algorithm. For vertices in H_i we ensure that the two conditions hold simultaneously: the number of edges lost is at least a half and, at the same time, for almost all of the vertices, it is not much less than that. For vertices in $N(H_i)$ on the other hand, we only care that the first requirement is satisfied. The third condition of the definition of a spanner states that all vertices, whether in H_i or $N(H_i)$, lose at least half of the edges at each iteration, so that after $k(B) := \lceil \log_2 D \rceil - 4$ many iterations, where $D := n/2^i$ is the upper bound on the maximum degree of vertices in H_i, the degree of every vertex in the spanner is at most 16.

In the next section we shall see how to compute $(\frac{1}{2}, 16)$-spanners in $O(\log^3 n)$ many rounds. In this section we assume that this can be done by means of a procedure Spanner. The algorithm for computing a matching in a block computes first a spanner in the block, and then a matching by means of a simple two step procedure– each vertex of the left-hand side proposes one edge and then every vertex of right-hand side selects one among the proposed edges incident to itself.

Definition 9 *The* proposal *graph of a spanner S, denoted by P, is any graph defined by the following procedure: every vertex $v \in l(S)$ selects one of its incident edges, arbitrarily. The graph P is the subgraph induced by the edges so selected.*

Vertices in $r(S)$ which have no proposed edge incident to them do not belong to P. Therefore, $l(P) = l(S)$ and $r(P) \subseteq r(S)$. The latter inclusion can be strict. Note that P is a set of "stars" with the high degree on the right side.

Definition 10 *The* matching *graph of a proposal graph P, denoted by M, is any graph defined by the following procedure: every vertex $v \in r(P)$ selects one of its incident edges, arbitrarily. The graph M is the subgraph induced by the edges so selected.*

Clearly M is a matching. Note also that $r(M) = r(P)$ and $l(M) \subseteq l(P)$. The last inclusion can be strict. To summarize, here is the algorithm for selecting a matching M in a block $B = (H, N(H), E_B)$. The blocks are generated at the outset, in constant time. For this to happen, every left-hand-side vertex needs to determine to which set H_i it belongs to. This takes constant time, provided that every vertex knows the value of n (or, equivalently, the value of Δ). Recall that a right-hand-side vertex might participate to many blocks.

Procedure MatchBlock

1. Let i be the input parameter. Compute a spanner S in B_i, the block defined by H_i, by invoking procedure SPANNER.
2. Compute a proposal graph $P \subseteq S$ as follows: every $v \in l(S)$ selects one incident edge arbitrarily. Vertices of $r(S)$ with no selected edges incident to them do not belong to P.
3. Compute a matching $M \subseteq P$ as follows: every vertex $u \in r(P)$ selects one incident edge arbitrarily.

The running time of procedure MATCHBLOCK is dominated by the number of rounds needed to compute a spanner– everything else takes constant time. Therefore, MATCHBLOCK takes

$$T(\text{MATCHBLOCK}, n) = T(\text{SPANNER}, n) = O(\log^3 n)$$

many rounds. The bound on the running time of SPANNER will be stablished in the next section.

After executing MATCHBLOCK in parallel in all blocks, we have a collection of matchings $\{M_i : 0 \leq i \leq \lfloor \log_2 n \rfloor\}$, with one matching in every block. Their union is not necessarily a matching, for vertices on the right-hand side might participate to more blocks and end up having degree higher than one. The final matching is determined by the following procedure:

Procedure GlobalMatch

Every vertex on the right-hand side selects, among the edges incident to itself, the edge of greatest weight $w_G(e)$.

To summarize, here is the overall algorithm for computing a matching that "touches" a constant fraction of the edges of a given two-coloured, bipartite graph (the proof of correctness will follow shortly). Given two graphs X and Y, $X \oplus Y$ is the graph whose vertex and edge sets are, respectively, $V(X) \cup V(Y)$ and $E(X) \cup E(Y)$.

Procedure BipartiteMatch

---

1. Generate a partition of $G = (L, R, E)$ into blocks, where block B_i is induced by the set H_i.
2. Compute a matching M_i inside every block B_i by invoking procedure MatchBlock.
3. Compute a matching $\mathcal{M}$ inside the graph $\oplus_i M_i$ by invoking procedure GlobalMatch.

The overall running time of the procedure is dominated by the time needed by procedure MatchBlock and is

$$T(\text{BipartiteMatch}, n) = T(\text{MatchBlock}, n) = O(\log^3 n)$$

The correctness of the procedure– that indeed $\mathcal{M}$ is a matching in G– is straightforward. The next two lemmas show that $touch(\mathcal{M})$ contains a constant fraction of the edges of G.

Lemma 11 *For every matching M computed by Step 2 of procedure* BipartiteMatch, *$touch(M)$ is a constant fraction of the edges of its block B.*

Proof Let $m := |E(B)|$ and

$$D := \max_{u \in l(B)} \deg_B(u).$$

Fix any $b \in (0, 1)$ and let us consider two cases: either

$$\sum_{v \in r(P)} d_B(v) \geq bm$$

or

$$\sum_{v \in r(P)} d_B(v) < bm.$$

For the first case, we know that $r(M) = r(P)$, therefore

$$|touch(M)| \geq bm.$$

For the second case, note that any proposal graph P satisfies the following conditions:

- $|l(P)| \geq a|l(B)|$
- For every $u \in l(P)$, $d_P(u) = 1$
- For every $v \in r(P)$, $d_P(v) \leq \frac{1}{2^{k(B)}} d_B(v) + 1$, where $k(B) := \lceil \log_2 D \rceil - 4$

and also

$$|l(B)| \frac{D}{2} \leq m \leq |l(B)| D$$

since B is a block. Therefore, we can bound the set of left-hand-side matched vertices as follows:

$$\begin{aligned}
|l(M)| &= |l(P)| - \sum_{v \in r(P)} (d_P(v) - 1) \\
&\geq a|l(B)| - \sum_{v \in r(P)} (d_P(v) - 1) \\
&\geq a|l(B)| - \sum_{v \in r(P)} \frac{1}{2^{k(B)}} d_B(v) \\
&\geq a|l(B)| - \frac{1}{2^{k(B)}} bm \\
&\geq a|l(B)| - \frac{1}{2^{k(B)}} bD|l(B)| \\
&\geq (a - 16b)|l(B)|
\end{aligned}$$

Recalling that for all $u \in l(B)$

$$\frac{D}{2} \leq d_B(u) \leq D$$

we get the bound

$$\begin{aligned}
|touch(M)| &\geq |l(M)|\frac{D}{2} \\
&\geq \frac{a - 16b}{2} D|l(B)| \\
&\geq \frac{a - 16b}{2} m
\end{aligned}$$

$\bigtriangledown$

Lemma 12 *The set touch($\mathcal{M}$), where $\mathcal{M}$ is the matching computed by Step 3 of Procedure* BipartiteMatch, *contains a constant fraction of edges of the input graph G.*

Proof This follows from two observations. Recall that we are operating inside a bipartite graph $G = (L, R, E)$ whose left-hand side L is partitioned by the sets H_i, which are defined as the sets of vertices u such that $n/2^{i+1} < d_G(u) \leq n/2^i$, for $i \geq 0$. It follows that for every set of indices $i < j_1 < j_2 < \ldots < j_m$ and no matter how we pick elements $u \in l(B_i) = H_i$ and $v_k \in H_{j_k}$,

$$d_G(u) \geq \frac{1}{2} \sum_{j \in J} d_G(v_j)$$

where $J := \{j_1, j_2, \ldots, j_m\}$. Note that in the above summation there is a unique, but arbitrary, v_j for every H_j. The second observation is that procedure GlobalMatch selects the edge of biggest weight $w_G(\cdot)$. $\bigtriangledown$

Therefore, we obtain the following lemma.

Lemma 13 *Procedure* BipartiteMatch, *given in input a two-coloured, bipartite graph G, produces a matching $\mathcal{M}$ in $O(\log^3 n)$ many communication rounds such that $|\mathrm{touch}(\mathcal{M})|$ is a constant fraction of the number of edges of G.*

5 How to Compute a Spanner

In this section we shall show how to compute an (a, d)-spanner in a given block $B = (H, N(H), E)$. The basic intuition was outlined in Section 4 but we repeat it here. The spanner is computed by approximately halving the degree of vertices of B. Recall that, by definition, in a block B_i the vertices of the left-hand side H_i have degree between $n/2^i$ and $n/2^{i+1}$. Consider the ideal situation in which the degree of every vertex is cut perfectly in half. Then, there exists a k, with $k = O(\log n)$, such that after k iterations, we have a subgraph of B in which every left-hand vertex has degree between, say, 1 and 4. Israeli and Shiloach accomplish this much in the PRAM model of computation by resorting to Euler circuits [10]. Unfortunately in our distributed model of computation the computation of Euler circuits requires a time proportional to the diameter of the network [25]. Therefore a completely different approach is called for.

We start with some preliminaries. Given a set of edges F, consider a decomposition of $G[F]$ into a collection of cycles and paths, computed as follows. Each vertex of $G[F]$, in parallel, splits itself into vertices of degree two (pairing any two adjacent edges) and perhaps one vertex of degree one (if the degree of the original vertex is odd). The new vertices are called **siblings** and the original vertex is called the **parent**. A new graph is obtained with at least $|F|$ vertices and exactly $|F|$ edges. Such a decomposition is called here a **2-decomposition**. Note that a 2-decomposition can be generated in constant time in the distributed model of computation and that it consists solely of paths and cycles.

Consider now the problem of halving the degree of a parent. If we could remove exactly one edge incident to every sibling, parents whose degree is even would lose exactly half of their edges, while parents of odd degree would have their degree split almost perfectly in half. A possible approach to achieve this could be the following. Compute perfect or near perfect matchings in the paths and cycles of the 2-decomposition and remove the edges so matched. In the sequel, we will refer to both perfect and near perfect matchings with the acronym **nPM**'s. Unfortunately computing nPM's in the a 2-decomposition requires linear time in the worst case [25]. To circumvent the problem, we "chop up" (partition) the 2-decomposition into smaller pieces and compute nPM's inside every piece, deleting the edges that are matched. This approach introduces errors because of "border" vertices that is, vertices that belong to two adjacent pieces. The problem arises because a border vertex may be matched or unmatched from each "side" at the same time. If we decide to remove only matched edges, a parent that has many siblings as border vertices will not lose enough edges. Conversely, if we decide to remove all edges incident on border vertices, a parent might lose too many of them. This conundrum can be solved by a judicious choice of the length of the pieces with which the 2-decomposition is partitioned. These pieces should be neither too long for efficiency reasons, nor too short, so that the number of border vertices can be bounded. If we can do this overall "most" parents will have their degree "almost perfectly" cut in half.

The above discussion motivates the following definitions. Given a partition of a 2-decomposition into connected components, the components are called **segments**. A segment is **long** if its length is at least

$$\ell := 100 \log^2 n$$

and **short** otherwise. The length of a segment is given by the number of edges it contains. A segment can only be an even cycle or a path. A vertex can possibly belong to two (adjacent) segments. Such vertices are called **border** vertices.

An edge adjacent to a border vertex is a **bad edge**. All other edges are **good**. Notice that edges adjacent to ends of paths are good.

Definition 14 *A parent v is called* **pliable** *if it is adjacent to at most $d(v)/p$ bad edges. The value of p, the coefficient of pliability, is*

$$p := \log n.$$

All parents that are not pliable are called **nasty**.

Remark: We can see now why it is necessary to operate inside bipartite graphs. In an odd cycle there is always going to be one sibling whose edges are unmatched. Conceivably the 2-decomposition could consist entirely of triangles and as a consequence almost all parents would be nasty, i.e., after removal of matched and bad edges they would lose all of their incident edges. In a bipartite graph on the other hand, cycles create no problems. When a cycle is short a perfect matching can be computed inside it, while when it is partitioned into long segments it cannot create too many nasty parents.

We are ready to present our main procedure, called SPANNER, together with subroutines SPLITTER and LONGARROWS. SPANNER acts on a block, defined by the input parameter D. The block is the two-coloured, bipartite graph $B = (H, N(H), E)$ defined by the set of left-hand vertices

$$H = \{u : \frac{D}{2} < \deg(u) \leq D\}.$$

Clearly, given D, vertices can decide whether they belong to H locally. A bird's eye view of the algorithm is as follows. Starting with $B^0 := B$ and $P_0 := l(B^0) = H$ the algorithm generates a 2-decomposition of the block B_0 and partitions it into long segments. After computing nPMs inside the segments, both matched and bad edges are marked (removed).

Let P_1, $P_1 \subseteq P_0$, be the set of pliable vertices. These are the vertices whose degree has been split well. Next, both marked edges and nasty vertices (the set $P_0 \setminus P_1$) are removed from B^0, yielding a new left-over block B^1. Then, a new 2-decomposition of B^1 is generated and partitioned into long segments. Again nPMs are computed inside the segments and both matched and bad edges are marked. This defines a new set $P_2 \subseteq P_1$ of vertices that have been pliable twice in a row, and so on. Therefore we obtain a sequence of subgraphs $B^0, B^1, \ldots, B^i$ and a sequence of subsets $P_0 \supseteq P_1 \supset \ldots \supseteq P_i$ of pliable vertices, where $P_j = l(B^j)$.

Let us denote the degree of vertex v in B^j by $d_j(v)$. We will show that after k stages, where $k = O(\log D)$,

(a) the size of P_k is a constant fraction of that of P_0, and

(b) for all $u \in P_k$, $d_k(u) = (1 \pm o(1))d_0(u)/2^k$.

Procedure SPANNER is as follows:

Procedure Spanner

---

1. Let $B^0 := B$, $P_0 := l(B^0)$.
2. For $j := 0$ to $k = O(\log D)$ do:
   (a) invoke procedure SPLITTER to mark edges of B^j;
   (b) each vertex of P_j, in parallel, enters the set P_{j+1} if it has less than $d_j(u)/p$ incident bad edges;
   (c) remove marked edges and vertices $P_j \setminus P_{j+1}$ from B^j, in order to get B^{j+1}.

The running time of Spanner is:

$$T(\text{Spanner}, n) = O(\log D \times T(\text{Splitter}, n)). \tag{1}$$

Procedure Splitter acts on B^j, the current subgraph, by computing nPMs in the segments of the 2-decomposition in B^j, and marking the resulting matched and bad edges. The segments, which are computed by procedure LongArrows (described below), will be directed: Each edge in the segment is given an orientation, and the orientation inside each segment is consistent, i.e. each segment is either a directed path or a directed cycle. The procedure to achieve this is described below. Here we notice that given an orientation in a 2-coloured graph, a nPM can be computed in constant time as follows: Let red and blue be the two colours; an edge enters the nPM if and only if its head points to a red vertex.

Procedure Splitter

---

1. Let B be the current subgraph. Each vertex of B, in parallel, generates its siblings, giving raise to a 2-decomposition of B.
2. Procedure LongArrows(ℓ) is invoked to compute oriented segments in the 2-decomposition.
3. In each segment a nPM is computed using the given 2-vertex-colouring of B as follows: Let red and blue be the colours of the bipartition. Then, each edge enters the matching if and only if its head points to a red vertex.
4. Edges incident to border vertices– i.e. vertices separating two adjacent segments– enter the set of bad edges
5. All matched and bad edges are marked.

The running time of Splitter is:

$$T(\text{Splitter}, n) = T(\text{LongArrows}, n). \tag{2}$$

Let us now turn to procedure LongArrows, the one responsible for partitioning the 2-decomposition into segments and computing nPM's inside them. We remark that such a partitioning could be computed by resorting to ruling-sets ([1]), but the procedure given here is both simpler and more efficient.

Initially each edge is given an arbitrary orientation. Each oriented edge is an **elementary arrow**. An **arrow** is a sequence of elementary arrows pointing to the same direction. Therefore adjacent arrows always have contradictory directions. Given a path or an even cycle P of length k, procedure LongArrows(ℓ) partitions P into arrows of length at least ℓ. More specifically, if $\ell \geq k$ the whole of P becomes an arrow of length k, otherwise P is partitioned into arrows of length at least ℓ. This can be done optimally in time $O(\ell)$.

We distinguish two types of borders between arrows: "head to head" and "tail to tail."

Procedure LongArrows

---

1. Let ℓ be the input parameter. At the beginning all elementary arrows are set arbitrarily.
2. For i:=1 to $\lceil \log \ell \rceil$ do
   In parallel, for every border of type "head to head" do:
   (a) if both arrows (touching the border) have length $< 2^i$ then choose one of them and reverse it (i.e. all of its elementary arrows are reversed)
   (b) If an arrow has length $< 2^i$ then reverse it

Note that we can check if an arrow has length $< 2^i$ in time $O(2^i)$ and we can reverse that arrow in time $O(2^i)$. It means that the whole algorithm takes

$$\sum_{i=1}^{\log \ell} O(2^i) = O(\ell)$$

rounds.

Fact 15 *Let C be a connected component of a 2-decomposition of a graph G. After* LongArrows*(ℓ) terminates either C is an arrow or it is partitioned into arrows of length at least ℓ.*

Proof Let k be the length of C and denote by $S(i)$ the sentence *"at the end of the i-th iteration all arrows have length $\geq 2^i$"*. $S(0)$ is true. Let us assume $S(i)$ and prove $S(i+1)$, for $i < \log k$. At the beginning of iteration $i+1$ there are only "long arrows" of length $\geq 2^{i+1}$ and "short arrows" of length $\geq 2^i$, $< 2^{i+1}$. We have to prove that at the end of the current iteration all arrows are long. Every "short arrow" has to be in one of the following situations:

(a) its head touches the head of another short or long arrow– in both cases it will disappear;

(b) its head touches the end of the path; in this case Step (2b) of procedure LongArrows will reverse it.

Therefore, after $\lceil \log \ell \rceil$ iterations either C is an arrow, if $k \leq 2\ell$, or it is partitioned into arrows of length at least ℓ.

$\bigtriangledown$

We now proceed to analize the behaviour of procedure Splitter. In the next two facts we shall describe the behaviour of degrees of pliable vertices and the cardinality of the set of such vertices in a single call of the subroutine Splitter. In the sequel we shall keep the same notation as in the above procedures and use the following notation: By Δ_j and δ_j we denote the maximum and minimum degree in B^j of vertices from $P_j = l(B^j)$. The first fact says that the degree of pliable vertices is split almost perfectly. Essentially, it establishes conditions 2 and 3 of the definition of (a,d)-spanner (the formal proof is given below in Theorem 18).

Fact 16 *For all $v \in P_{j+1}$,*

$$\frac{1}{2}\left(\left(1 - \frac{2}{\log n}\right) d_j(v) - 1\right) \leq d_{j+1}(v) \leq \frac{1}{2}\ \left(d_j(v) + 1\right),$$

where j is any iteration of the for-loop of procedure Spanner. *The right-hand-side inequality holds for all $v \in V(B^j)$.*

Proof Let e_+ and e_- denote the number of good and bad edges incident to a parent v, respectively. To bound $d_j(v)$ from above, note that the worst case occurs when v has no bad edges incident to itself. Therefore, when the $d_j(v)$ is odd and its unique degree-1 sibling has no marked edge,

$$d_{j+1}(v) \leq \frac{1}{2}(e_+ + 1) = \frac{1}{2}(d_j(v) + 1)$$

This holds for all $v \in V(B^j)$.

For the lower bound, notice that a pliable parent v looses the largest number of edges if all of its siblings incident to bad edges have just one bad edge incident to them and have the other edge marked, so that both edges will be lost. Then, when v has odd degree,

$$d_{j+1}(v) \geq \frac{1}{2}(e_+ - e_- - 1) \geq \frac{1}{2}\left(\left(1 - \frac{2}{\log n}\right) d_j(v) - 1\right),$$

since $e_- \leq d_j(v)/\log n$ by definition of pliable parent. $\bigtriangledown$

The next fact says that most vertices remain pliable from one iteration to the next, thereby establishing condition 1 of the definition of (a,d)-spanner (again, the formal proof of this claim is given in Theorem 18 below).

Fact 17 *For all iterations $j = 0,\ldots,k-1$ of procedure* SPANNER*,*

$$|P_{j+1}| \geq |P_j|\Big(1 - \frac{2/100}{\log n}\frac{\Delta_j}{\delta_j}\Big)$$

Proof Let N_{j+1} be the set of nasty vertices at the end of iteration j and let $be[N_{j+1}]$ be the number of bad edges incident to N_{j+1}. Notice that $|P_{j+1}| = |P_j| - |N_{j+1}|$.

A lower bound for $be[N_{j+1}]$ follows from the fact that vertices in N_{j+1} have, by definition, at least $\delta_j/\log n$ bad edges. Hence,

$$be[N_{j+1}] \geq |N_{j+1}|\frac{\delta_j}{\log n}.$$

Recall that our graph is bipartite and so all cycles (and in particular the short ones) of the 2-decomposition are of even length. Therefore all bad edges arise from nPMs in long components (paths and cycles) only.

We can bound the number of bad edges incident to N_{j+1} by the total number of bad edges in B^j. Since $|E(B^j)| \leq \Delta_j|P_j|$, and since each nPM has length at least $\ell = 100\log^2 n$ and contributes at most two bad edges,

$$be[N_{j+1}] \leq \frac{2|P_j|\Delta_j}{100\log^2 n}$$

and the fact follows. $\bigtriangledown$

Finally, we shall present the analysis of procedure SPANNER.

Theorem 18 *An invocation of procedure* SPANNER *with parameter D and with input graph $G = (V,E)$ computes an $(\frac{1}{2},16)$-spanner w.r.t to G and the set $H := \{v \in V(G) : D/2 \leq \deg_G(v) \leq D\}$.*

Proof We shall check whether the output graph of our procedure fulfills the three conditions it has to satisfy to be a spanner (Definition 8); that is, for $k := \lceil \log D \rceil - 4$,

- $|P_k| \geq \frac{1}{2}|P_0|$;
- for every $u \in P_k$, $d_k(u) \in [1,16]$;
- for every $v \in r(B^k)$, $d_k(v) \leq 2^{-k}d_0(v) + 1$.

Fix $k \geq 1$ and note that $D/2 \leq d_0(v) \leq D$. By an easy induction, using Fact 16, we have that, for all $v \in P_k$,

$$q^k\left(\frac{D}{2}+1\right) - 1 \leq d_k(v) \leq \left(\frac{1}{2}\right)^k (D-1) + 1,$$

where $q = (1 - 2/\log n)/2$. The upper bound, that holds also for $v \in r(B^k)$, establishes condition 3 of the definition of a spanner. For condition 2, if we set $k = \lceil \log D \rceil - c$, we obtain

$$q^k \left(\frac{D}{2} + 1\right) - 1 > 2^{c-1} e^{-2(1+\frac{2}{\log n})} - 1,$$

and

$$\left(\frac{1}{2}\right)^k (D-1) + 1 < 2^c + 1$$

Hence, choosing $c = 4$, vertices from the set P_k all have degrees belonging to the interval $[1, 16]$, for large enough n. This establishes the second condition.

Finally, we have to show that the first condition which determines the spanner also holds. That is,

$$|P_k| \geq \frac{1}{2}\,|P_0|.$$

Applying repeatedly the inequality established in Fact 17, we get

$$|P_k| \geq |P_0| \prod_{j=0}^{k-1} \Big(1 - \frac{2/100}{\log n} \frac{\Delta_j}{\delta_j}\Big).$$

However

$$\frac{\Delta_j}{\delta_j} \leq \frac{2^{-j}(D-1)+1}{q^j(D/2+1)-1} \leq 16,$$

since the last fraction is an increasing function of j, for $j \leq k$. Therefore, for n large enough,

$$|P_k| \geq |P_0| \Big(1 - \frac{32/100}{\log n}\Big)^{\log n} \geq e^{-33/100} |P_0| \geq \frac{1}{2} |P_0|$$

$\bigtriangledown$

Acknowledgements

The third author gratefully acknowledges the hospitality of the Adam Mickiewicz University, where much of this work was done, and the financial support of the Alexander von Humboldt Foundation.

References

[1] B. Awerbuch, A.V. Goldberg, M. Luby, and S. Plotkin, Network decomposition and locality in distributed computing, in Proceedings of the 30th Symposium on Foundations of Computer Science (FOCS 1989), pages 364-369, IEEE, Research Triangle Park, North Carolina.

[2] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg, Fast network decompositions, in Proceedings of the 1992 ACM Symposium on Principles of Distributed Computing (PODC 92), pp.169-177

[3] N. Alon, J. Spencer, and P. Erdős, The Probabilistic Method, Wiley–Interscience Series, John Wiley & Sons, Inc., New York, 1992.

[4] B. Bollobás, Graph Theory, Springer Verlag, New York, 1979.

[5] B. Bollobás, Chromatic number, girth, and maximal degree, *Discrete Math..* **24** (1978), 311–314.

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

[7] A.V. Goldberg, S.A. Plotkin and G.E. Shannon, Parallel symmetry-breaking in sparse graphs, SIAM J. Disc. Math. Vol.1, No. 4, November 1988, pp. 434-446

[8] D.A. Grable and A. Panconesi, Nearly optimal distributed edge colouring in $O(\log \log n)$ rounds, Random Structures and Algorithms, 10(3):385-405, May 1997. Preliminary version in Proceedings of the Eight Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 97), New Orleans.

[9] M. Hańćkowiak, M. Karoński and A. Panconesi, On the distributed complexity of computing maximal matchings, in Proceedings of SODA 98, the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, pp. 219-225.

[10] A. Israeli and Y. Shiloach, An improved algorithm for maximal matching, Information Processing Letters, 22(2):57-60, 18 January 1986

[11] Yijie Han, An improvement on parallel computation of a maximal matching, Information Processing Letters, 56 (1995), pp. 343-348

[12] H.J. Karloff and D.B. Shmoys, Efficient parallel algorithms for edge coloring problems, J. Algorithms, 8 (1987), pp. 39-52

[13] R. M. Karp, Probabilistic recurrence relations, in proceedings of the 23rd Annual ACM Symposium on Theory of Computing (STOC 91), pages 190–197, New Orleans, Louisiana.

[14] S. Kutten and D. Peleg, Fast distributed construction of k-dominating sets and applications, in Proceedings of the 1995 ACM Symposium on Principles of Distributed Computing (PODC 95), Ottawa, Ontario, pp. 238–249

[15] N. Lynch, Distributed Algorithms, Morgan-Kaufmann, San Francisco.

[16] Lecture Notes in Distributed Algorithms, Solution Sheet # 4, Available at http://www.nada.kth.se/kurser/kth/2D5340.

[17] N. Linial, Locality in distributed graph algorithms. SIAM Journal on Computing, 21(1):193-201, February 1992.

[18] N. Linial and M. Saks, Low diameter graph decomposition, Combinatorica (1993), Vol. 13 (4)

[19] M. Luby, A simple parallel algorithm for the maximal independent set problem. In Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC 85), pages 1-10, Providence, Rhode Islands.

[20] M. Luby, Removing randomness in parallel without processor penalty, Journal of Computer and System Sciences, 47(2):250-286, October 1993

[21] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[22] Moni Naor, A lower bound on probabilistic algorithms for distributive ring coloring, SIAM J. Disc. Math., Vol. 4, No. 3, pp. 409-412, August 1991

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

[24] Ö. Johansson, personal communication.

[25] A. Panconesi, Lecture Notes in (Theoretical) Distributed Computing, KTH NADA Tech Reprt TRITA-NA-9801

[26] A. Panconesi and A. Srinivasan, The Local Nature of Δ-coloring and Its Algorithmic Applications, Combinatorica 15 (2) 1995, 255-280.

[27] A. Panconesi and A. Srinivasan, On the complexity of Distributed Network Decomposition, Journal of Algorithms 20, 356–374 (1996).