A Faster Distributed Algorithm for Computing Maximal Matchings Deterministically (Extended Abstract)

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
Dept of CS
University of Bologna
Bologna, 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 improves on an earlier result by three log-factors.

1 Introduction

In this paper we show that maximal matchings (MM's) can be computed deterministically in $O(\log^4 n)$ rounds in the synchronous, message-passing model of computation. This improves substantially on an earlier result by the present authors, which shows that MM's can be computed in $O(\log^7 n)$ many rounds [9]. This rather substantial improvement in asymptotics is based on several new algorithmic ideas that, we hope, might prove useful in other contexts. 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 which 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 [16] 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 [6, 21].

Generalizing from rings to bounded degree graphs one sees that several classical graph structures of both theoretical and practical interest, including MIS's, maximal matchings, $(\Delta+1)$- and even Δ-vertex colorings, can be computed in poly-logarithmic time [1, 2, 7, 25]. 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, 17, 22, 25, 26]. 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 poly-logarithmically many rounds. Furthermore, these randomized algorithms are usually extremely simple and their actual complexity is very low. For instance, $(\Delta+1)$–vertex coloring and MIS can be computed in $O(\log n)$ rounds with high probability by exceedingly simple protocols [18, 19, 23]. Another important case is that of $(O(\log n), O(\log n))$–decompositions, a very interesting type of graph decomposition with many applications, which can be computed in $O(\log^2 n)$ rounds [17]. 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 which run in polylogarithmically many rounds in the size of the network.

In an attempt to gain some insight into this problem, the present authors studied the problem of computing maximal matchings and showed that, in this case, randomization is not needed [9]. In this paper we present a new solution which, asymptotically, improves substantially over the old one– $O(\log^4 n)$ rounds versus $O(\log^7 n)$ rounds. Although the time complexity is still rather 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 the same $O(\log^4 n)$ [10]. The improvement in asymptotics is owed to some new algorithmic ideas that could be useful in other contexts.

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.

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 assumption on the input network. Other notable exceptions are the so-called ruling forests of [1] and the k-dominating sets of [13] 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. We also need the notion of (α,β)*-ruling sets* [6]. An (α,β)-ruling set with respect to a graph $G=(V,E)$ and a subset of nodes $P \subseteq V$, is a set of vertices S such that: (a) $S \subseteq P$; (b) any two vertices of S are at distance (in G) at least α from each other, and (c) every vertex of P is at distance (in G) at most β from some veretx of S. In [1] the following result is proven.

Theorem 1 . *For all positive integers $k > 0$, $(k, k\log n)$-ruling sets can be computed in $O(k \log n)$ rounds in the synchronous, message-passing model of computation.*

Given a degree-2 graph C (i.e. a collection of paths and cycles), a **subdivision** of C is simply a partition of the edges of C into segments of length at least 2 and at most $O(\log n)$. For the things to follow, it is important to note that adjacent segments are vertex disjoint, but share a common edge. The length of a segment is the number of vertices in it. The following corollary is an easy consequence of Theorem 1.

Corollary 2 *Subdivisions of degree-2 graphs can be computed in $O(\log n)$ rounds in the synchronous, message-passing model of computation.*

The easy proof is omitted from this extended abstract.

3 Reducing the problem to two-colored 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. In what follows, and throughout the rest of the paper, given a set of edges X, $match(X)$ denotes the set of edges matched by X, i.e. the set of edges of G incident to at least one edge from X. By convention, $X \subseteq match(X)$. 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 $|match(\mathcal{M})|$ 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 on them. The left-over graph is given in input to the next phase. This takes $O(\log n)$ phases since, by assumption, $|match(\mathcal{M})|$ 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 which every vertex splits itself into two vertices (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 are 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 whole 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 $|match(\mathcal{M})|$ is a constant fraction of $|match(\mathcal{L})|$, 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 by "merging" each pair of siblings back together. C is a collection of paths and cycles and its edge set is exactly $\mathcal{L}$. The paths and cycles of C, potentially very long, are sub-divided into smaller, more manageable segments (which are themselves paths or cycles) of length at least two and at most $O(\log n)$. By Corollary 2 this can be quickly accomplished. Now, if a segment has an even number of vertices, we compute a perfect matching inside it. Otherwise, we opt for a simple greedy strategy: compute a near perfect matching, leaving out the extreme point of smaller degree or, for segments which are odd cycles, the vertex of smallest degree. Since the segments are vertex-disjoint, the perfect and near-perfect matchings computed inside each segment can be put together to form a true matching $\mathcal{M}$. This ends the phase. A simple counting argument shows that $\mathcal{M}$ matches a constant fraction of $|match(\mathcal{L})|$, 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 whole graph.

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

PROCEDURE MATCH

- $\mathcal{F} \leftarrow \emptyset$ [$\mathcal{F}$ will be the final matching]
- Repeat $O(\log n)$ times:

  1. Direct every edge arbitrarily;
  2. Every vertex u splits into two siblings u_{in} and u_{out}. All incoming edges of u become (non-oriented) edges incident on u_{in}, while all outgoing edges of u become (non-oriented) edges incident on 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 graph C of maximum degree 2 as follows: Each pair of siblings u_{in} and u_{out} is merged back into u. Edges of $\mathcal{L}$ which were incident on u_{in} or u_{out} become edges incident onto u.
  5. Compute a subdivision of C into segments of length at least 2 and at most $\log n$ (the length is the number of vertices). Inside each segment, if the number of vertices is even, compute a perfect matching; otherwise, if the number is odd and the segment is a path, compute a near-perfect matching by leaving out the extreme point of smaller degree or, if the segment is an odd cycle, by leaving out the vertex of smallest degree.
  6. Let $\mathcal{M}$ be the union of all matchings and perfect-matchings so computed. Add its edges 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.

Notice that the vertices must know n in order to run the algorithm.

The subdivision of C consists of a collection of segments (paths or cycles) which are vertex-disjoint. Therefore both $\mathcal{M}$ and $\mathcal{F}$ are matchings and, by construction, $\mathcal{F}$ is maximal when the algorithm terminates. It remains to show that $|match(\mathcal{M})|$ is indeed a constant fraction of the number of edges in the input graph, thereby ensuring termination in $O(\log n)$ phases. We do this next.

Lemma 3 *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 B such that $|match(\mathcal{L})|$ is constant fraction of $|E(\mathcal{B})|$. Then, $\mathcal{M}$ matches a constant fraction of the edges of G.*

Proof It is apparent from the algorithm that we can identify the two edge sets $E(G)$ and $E(\mathcal{B})$ (Steps 1 and 2), and the two edge sets $\mathcal{L}$ and $E(C)$ (Step 4). C is a collection of paths and cycles– every vertex of C has degree at most 2– and it is partitioned into a collection of vertex-disjoint segments $\mathcal{S}$, each having between 2 and $\log n$ vertices (Step 5). Let

$$\ell := |match(C)| = |match(\mathcal{L})|$$

and

$$s := \sum_{S \in \mathcal{S}} match(S).$$

Now, since every edge can be incident on at most two segments,

$$s \geq \ell \geq \frac{s}{2}.$$

Let

$$m := |match(\mathcal{M})|$$

and let

$$s' := \sum_{S \in \mathcal{S}} match(S \cap \mathcal{M})$$

i.e. the sum, over all segments, of the edges incident on the matching, including the matching itself, which is computed inside each segment. Again, s' might overcount edges, but every edge is counted at most twice. Therefore,

$$s' \geq m \geq \frac{s'}{2}.$$

and

$$s' \geq \frac{1}{2}s.$$

The last inequality follows from Step 5 of procedure MATCH. Since $E(\mathcal{B}) = E(G)$ and $\mathcal{L}$, by hypothesis, matches a constant fraction of edges of G,

$$\ell = \alpha|E(G)|$$

for some fixed constant $\alpha > 0$. Putting together the previous inequalities we obtain,

$$m \geq \frac{1}{2}s' \geq \frac{1}{2} \cdot \frac{1}{2}s \geq \frac{1}{4}\ell = \frac{\alpha}{4}|E(G)|.$$

$\bigtriangledown$

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 overall; by the results cited in Section 2 a subdivision can be computed in $O(\log n)$ many rounds and, after that is computed, the matchings inside each segment can also be computed in $O(\log n)$ many rounds by resorting to total communication. 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 4 *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-colored 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, 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 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 5 $H_i = \{u \in L : \frac{\Delta(G)}{2^i} \leq d(u) \leq \frac{\Delta(G)}{2^{i+1}}\}$.

Notice that the H_i's, for $0 \leq i \leq \lceil \log_2 \Delta(G) \rceil$, partition the set L of left-hand-side vertices.

Definition 6 *The* block *induced by H_i, denoted as B_i, is the subgraph of G induced by the edges incident on H_i. The set of neighbours of H_i in the block is denoted by $N(H_i)$. Obviosuly, $N(H_i) \subseteq R$.*

The blocks B_i's, for $0 \leq i \leq \lceil \log_2 \Delta(G) \rceil$, partition the edge set $E(G)$ and, obviously, the set of vertices L. Notice that a partition of G into blocks can be generated in constant time, provided that the value of $\Delta(G)$ or, alternatively, that of n is known. Two different blocks, say, B_i and B_j, can have righ-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 to 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 on 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 on 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. This simple locally greedy approach works– i.e. it results in matching a constant fraction of edges overall– because of the way the sets H_i are defined. Namely, the smallest degree vertex of H_i is at least twice that of the highest degree vertex of H_{i+2}. 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 $\ell(X) := V(X) \cap L$ 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 7 *A* (a, d)-spanner *of a block*

$$B = (H, N(H), E_B)$$

is a subgraph $S \subseteq B$ such that:

- $|\ell(S)| \geq a|H|$, *i.e. S contains a constant fraction of the vertices of H.*
- *For every $u \in \ell(S)$, $d_S(u) \in [1, d]$, i.e. the degree of left-hand-side vertices is constant; and,*
- *For every vertex $v \in r(S)$, $d_S(v) \leq 2^{-k}d_B(v) + 1$, where $k := \lceil \log_2 D \rceil$ and D is the maximum degree inside B among the vertices in $\ell(B)$, i.e. $D := \max_{u \in \ell(B)} d_B(u)$.*

Intuitively, a spanner is a subgraph of constant degree that spans a "large" fraction of vertices in H. Its structure makes it easy to compute a matching inside it which is incident on a constant fraction of edges in the block. At this point, the third condition might look quite contrived, so that a few words of comment are in order. Roughly speaking, a spanner is computed starting from a block by repeatedly slashing the vertex degrees. In the process we want to ensure that "almost all" vertices of H have their degree "almost perfectly" cut in half at each iteration of the algorithm. The third condition in the definition states that all vertices, whether in H or not, loose at least half of the edges at each iteration. In the next section we shall see how to compute $(e^{-65}, 16)$-spanners in $O(\log^3 n)$ many rounds.

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– the right-hand side proposes and the left-hand side selects among proposed edges incident on the same vertex.

Definition 8 *The* proposal *graph of a spanner S, denoted by P, is any graph defined by the following procedure: every vertex $v \in r(S)$ selects one of its incident edges, arbitrarily. The graph P is the subgraph induced by the edges so selected. Vertices in $\ell(S)$ which have no proposed edge incident on them do not belong to P. Therefore, $r(P) = r(S)$ and $\ell(P) \subseteq \ell(S)$. The latter inclusion can be strict.*

Definition 9 *The* weight *of an edge $e = uv$ in a graph X is defined as $w_X(e) := d_X(u) + d_X(v)$.*

Definition 10 *The* matching *graph of a proposal graph P, denoted by M, is any graph defined by the following procedure: every vertex $v \in \ell(P)$ selects the incident edge of bigger weight $w_B(e)$, breaking ties arbitrarily. The graph M is the subgraph induced by the edges so selected.*

Clearly M is a matching. Note also that $\ell(M) = \ell(P)$ and $r(M) \subseteq r(P)$. 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 Δ or, equivalently, the value of an upper bound such as n. Recall that a right-hand-side vertex might participate to many blocks.

Procedure MatchBlock

- Compute a spanner S in B.
- Compute a proposal graph $P \subseteq S$ as follows: every $v \in r(S)$ selects one incident edge arbitrarily. Vertices of $\ell(S)$ with no selected edges incident on them do not belong to P.
- Compute a matching $M \subseteq P$ as follows: every vertex $u \in \ell(P)$ selects the edge of maximum weight $w_B(e)$ among those incident upon itself, breaking ties 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

$$T(\textsc{MatchBlock}, n) = T(\textsc{Spanner}, n) = O(\log^3 n)$$

many rounds.

After executing MatchBlock in parallel in all blocks, we have a collection of matchings $\{M_i : 1 \leq i \leq \lceil \log_2 \Delta(G) \rceil\}$, 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 on itself, the edge of greatest weight $w_G(e)$.

Notice that we use $w_B(\cdot)$ in procedure MatchBlock, while $w_G(\cdot)$ is used in procedure GlobalMatch. To summarize, here is the overall algorithm for computing a matching that matches 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

- Generate a partition of $G = (L, R, E)$ into blocks, where block B_i is induced by the set H_i.
- Compute a matching M_i inside every block B_i by invoking procedure MatchBlock.
- 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

$$\begin{aligned} T(\textsc{BipartiteMatch}, n) &= T(\textsc{MatchBlock}, n) \\ &= = O(\log^3 n). \end{aligned}$$

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

Lemma 11 *Every matching M matches a constant fraction of the edges of its block B.*

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

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

Moreover, let b be any constant $0 < b < a$, where a is the constant in the definition of an (a, d)-spanner S (for definiteness and without loss of generality we can assume that $b := \frac{a}{2}$). We consider two cases: either

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

or

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

Let's begin with the first case. By definition, in every block B,

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

Recalling Definitions 6–10, we can bound the set of left-hand-side, matched vertices as follows.

$$\begin{aligned}
|\ell(M)| &\geq |\ell(S)| - \sum_{v \in r(S)} (d_S(v) - 1) \\
&\geq a|\ell(B)| - \sum_{v \in r(S)} (d_S(v) - 1) \\
&\geq a|\ell(B)| - \frac{1}{2^k} \sum_{v \in r(S)} d_B(v) \\
&\geq a|\ell(B)| - \frac{1}{2^k} bm \\
&\geq a|\ell(B)| - \frac{1}{2^k} bD|\ell(B)| \\
&\geq (a-b)|\ell(B)|.
\end{aligned}$$

Recalling the definition of a block we see that

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

for every vertex $u \in \ell(B)$. Therefore, the number of matched edges in this case can be bounded as follows

$$\begin{aligned}
|match(M)| &= \sum_{e=uv \in M, u \in L} d_B(u) \\
&\geq |\ell(M)| \frac{D}{2} \\
&\geq \frac{a-b}{2} D |\ell(B)| \\
&\geq \frac{a-b}{2} m.
\end{aligned}$$

For the other case, recalling the definition of spanner and the fact that in Definition 10 the edge of biggest weight $w_b(\cdot)$ is chosen,

$$\begin{aligned}
|match(M)| &= \sum_{e=uv \in M, v \in R} d_B(v) \\
&\geq \frac{1}{d} \sum_{v \in r(P)} d_B(v) \\
&= \frac{1}{d} \sum_{v \in r(S)} d_B(v) \\
&\geq \frac{b}{d} m
\end{aligned}$$

$\bigtriangledown$

Lemma 12 *The matching $\mathcal{M}$ matches a constant fraction of edges in the 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 $\Delta(G)/2^{i+1} \leq d_G(u) \leq \Delta(G)/2^i$. It follows that for every set of indices $i < j_1 < j_2 < \ldots < j_m$ and no matter how we pick elements $u \in \ell(B_i) = H_i$ 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-colored, bipartite graph G, produces a matching $\mathcal{M}$ in $O(\log^3 n)$ many communication rounds that matches a constant fraction of the 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. Consider the ideal situation in which the degree of every vertex is cut perfectly in half. Then, in $O(\log n)$ iterations we have a subgraph of B in which every left-hand vertex has degree between, say, 1 and 4 (recall the definition of a block B_i, where vertices of H_i have degree between $\Delta/2^i$ and $\Delta/2^{i+1}$). Israeli and Shiloach accomplish this much in the PRAM model of computation by resorting to euler circuits[10]. It can be shown however, that in our distributed model the computation of Euler circuits requires a time proportional to the diameter of the network [24]. This calls for a completely different approach, explained below.

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**. So, we get a new graph with at least $|F|$ vertices and exactly $|F|$ edges and such decomposition is called here a **2-decomposition**. Note that this can be done in constant time in the distributed model of computations.

We classify all components of a 2-decomposition into two groups. A component is called **long** if its length is at least $\ell := \log^2 n$ and **short** otherwise. A connected subgraph of a 2-decomposition is called a **segment**.

Consider the problem of halving the degree of a parent. This could be done by removing exactly one edge incident on each sibling. Except for an extra edge in case of parents of odd degree this would cut the degree perfectly in half. To remove edges incident on siblings we could 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. For sake of clarity we will also speak of **marked** edges as opposed to edges matched or removed in the 2-decomposition. Unfortunately computing nPM's in the whole 2-decomposition requires, in the worst case, linear time [24]. To circumvent the problem, we "chop up" (partition) the 2-decomposition into smaller segments that by virtue of the fact that they are two-coloured or because of the fact that they are short, allow nPM's to be easily computed inside them. Following the plan, the matched edges are marked, i.e removed from the graph. This introduces some errors– not all parents will have their degree halved– but, if the the segments are long enough, overall "most" parents will have their degree "almost perfectly" cut in half.

One way to do the partitioning would be to resort to ruling-sets [1]. This would allow to partition the 2-decomposition into segments of length at least ℓ and at most $\ell \log n$ in $O(\ell \log n)$ rounds. But we can do better. Taking advantage of the fact that we are working inside a bipartite graph with a given 2-coloring, we can find a partition into segments of length at least ℓ *and* compute near perfect matching inside them. This is achieved through a procedure called LongArrows which is explained below. The 2-colouring of the 2-decomposition is that induced by the parent graph– siblings inherit the colour of their parent. (Recall that we are working inside a bipartite graph and that each parent knows which side of th ebipartition it belongs to).

To summarize, we partition the 2-decomposition into segments and compute a nPM inside each segment. Two adjacent segments share a common **border** vertex. The matched edges are marked (thrown away). The problem arises because of these border vertices, which may be matched or unmatched from each "side" of the segment at the same time. Suppose that we remove only edges of the nPM's. But then, if a parent has, say, all of its siblings as border vertices, then none of its edges will be marked and, as a result, none of them will be removed. The opposite approach– remove all edges incident on border siblings whether or not they are marked– might result in some parents losing lots or even all of their edges. Fortunately, as we shall see, this is not a problem because, if one chooses the segment length judiciously, this will happen to "very few" parents.

The discussion motivates the following definitions. We call an edge adjacent to a border vertex a **bad edge**. Otherwise an edge is called **good**. Notice that edges adjacent to ends of paths are good. 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 which 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 both edges are unmatched. In a distributed model we have no control over the way a 2-decomposition is going to look; it is possible that this be made of many short cycles, say, triangles, so that that "lots" of parents will have "lots" of bad siblings or, using the above terminology, that "lots" of parents will be nasty. On the other hand, short even cycles are great since a perfect matching can be computed in them, and long even cycles and paths are also good, since only border siblings can create trouble.

We are ready to present our main procedure, called Spanner, together with subroutines Splitter and LongArrows. Spanner acts on a block which is 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

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

Clearly, given D, vertices can decide whether they belong to H locally. A bird's eye view of the algorithm is as follows. We shall use throughout the following notation. Given a set of vertices X, the graph $G[in(X)]$ is the graph induced by the edges incident upon H. With this notation $B = G[in(H)]$. Starting from $P_0 := H$, the algorithm computes nPMs in the segments of the 2-decomposition of $G[in(P_0)]$. Afterwards both matched and bad edges are removed (marked). This defines a new set P_1 of pliable vertices which, intuitively, are those whose degree has been split well. Then, a new 2-decomposition of $G[in(P_1)]$ is produced and again nPM's are computed in its segments and thrown away (marked) together with the new set of bad edges, and so on. This defines a sequence $P_0 \supseteq P_1 \supset \ldots \supseteq P_i$ of sets of pliable vertices. We will show that after $k = O(\log D)$ stages:

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

(b) the degree of each vertex u in P_k is $(1 \pm o(1))d(u)/2^k$, where $d(u)$ is the degree of u prior to the invocation of Spanner.

Procedure Spanner is as follows:

Procedure Spanner(D)

1. Let $P_0 := \{u : D/2 \leq d(u) \leq D\}$. Let $d_0(u) := d(u)$ for each vertex in P_0.
2. For $j := 0$ to $k = O(\log D)$ do:
   (a) invoke procedure Splitter to mark edges of $G[in(P_j)]$;
   (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. Let $d_{j+1}(u)$ be the number of unmarked edges incident on u.

The running time of Splitter is:

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

Procedure Splitter acts on P_j, the current set of pliable vertices, by computing nPM's in the segments of the 2-decomposition of $G[in(P_j)]$, 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-colored 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 P be the current set of pliable vertices. Each vertex of P, in parallel, generates its siblings, giving raise to a 2-decomposition of $G[in(P)]$;
2. Procedure LONGARROWS(ℓ) is invoked to compute oriented segments of length at least $\ell = \log^2 n$ in the 2-decomposition
3. In each segment a nPM of cardinality at least ℓ is computed using the given 2-vertex-colouring of $G[in(P)]$ 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 on 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 SPANNER is:

$$T(\text{SPLITTER}, n) = T(\text{LONGARROWS}, n). \qquad (2)$$

We now show how segments can be given a consistent orientation. Each vertex has an **elementary arrow** indicating one of its neighbours. Elementary arrows are determined arbitrarily. An **arrow** is a sequence of elementary arrows indicating the same direction. Therefore adjacent arrows always have contradictory directions. We can compute arrows of length at least ℓ in time $O(\ell)$ with procedure LONGARROWS(ℓ) - and this algorithm is optimal !

The next procedure, on input ℓ, computes arrows that have length at least ℓ. Connected components of length greater than ℓ might end up split into arrows of length at least ℓ while components whose length is smaller than ℓ will be given a consistent orientation. We distinguish two types of borders between arrows: "head to head" and "tail to tail."

Procedure LONGARROWS(ℓ)

1. At the beginning all elementary arrows are arbitrary
2. For i:=1 to $\log \ell$ 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 14 *After* LONGARROWS *terminates there are only arrows of length $\geq \ell$.*

Proof Denote by $T(i)$ the sentence "at the end of the i-th iteration all arrows have length $\geq 2^i$."

$T(0)$ is true. Let's assume $T(i)$. We have to prove $T(i+1)$. 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 $\log \ell$ iterations all arrows have length $\geq T(\log \ell) = \ell$. $\bigtriangledown$

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 G_j we denote the graph $G[in(P_j)]$, and by Δ_j and δ_j the maximum and minimum degree of vertices in P_j (we do not care about the min degree of vertices $V(G_j) - P_j$ and, as shown by the next fact, their max degree is upper bounded by Δ_j).

The degree of $u \in G_j$ is denoted by $d_j(u)$. 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 17).

Fact 15 *For all $v \in P_j$,*

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

and

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

where j is any iteration of the for-loop of procedure SPANNER *(Note: the inequality does not fit in a line, so we split it). The right-hand-side inequality holds for all $v \in V(G_j)$.*

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

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

This holds for all $v \in V(G_j)$.

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

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

since $e_- \leq d_{j-1}(v)/\log n$ 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 17 below).

Fact 16 *For all iterations $j = 1, \ldots, k$ of procedure* Spanner*,*

$$|P_j| \geq |P_{j-1}|\left(1 - \frac{4}{\log n}\frac{\Delta_{j-1}}{\delta_{j-1}}\right)$$

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

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

$$be[N_j] \geq |N_j|\frac{\delta_{j-1}}{2\log n}.$$

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 on N_j by the the total number of bad edges in G_{j-1}. Since $|E(G_{j-1})| \leq \Delta_{j-1}|P_{j-1}|$, and since each nPM has length at least $\ell = \log^2 n$ and contributes at most two bad edges,

$$be[N_j] \leq \frac{2|P_{j-1}|\Delta_{j-1}}{\log^2 n}$$

and the fact follows. $\bigtriangledown$

Finally, we shall present the analysis of procedure Spanner.

Theorem 17 *An invocation of procedure* Spanner *with parameter D with input grapg $G = (V, E)$ computes an $(a, 16)$-spanner w.r.t to G and the set $H := \{v \in V(G) : D/2 \leq \deg_G(v) \leq D\}$, where $a \geq e^{-65}$.*

Proof We shall check whether the output graph of our procedure fulfills the three conditions it has to satisfy to be a spanner (see Definition 7).

Fix $k \geq 1$ and note that $D/2 \leq d_0(v) \leq D$. By an easy induction, using Fact 15, 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 establishes condition 3 of the definition of a spanner. For condition 2, if we set $k = \log D - c$, 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 in Definition 7.

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

$$|P_k| \geq a\ |H|.$$

Applying repeatedly the inequality established in Fact 16, we get

$$|P_k| \geq |H|\prod_{j=0}^{k-1}\left(1 - \frac{4}{\log n}\frac{\Delta_j}{\delta_j}\right).$$

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,

$$|P_k| \geq |H|\left(1 - \frac{64}{\log n}\right)^{\log n} \geq e^{-65}|H|$$

which gives a value of $a \geq e^{-65}$. $\bigtriangledown$

Remark: The constant a in Theorem 17 is extremely tiny. It can, however, be made very large at the expense of an extra log-factor in the overall complexity of the algorithm. If we use segments of length $\log^3 n$ instead of $\log^2 n$ the last inequality of Theorem 17 would become

$$\begin{aligned} |P_k| \geq |H|\left(1 - \frac{64}{\log^2 n}\right)^{\log n} &\approx \left(1 - \frac{64}{\log n}\right) \\ &= (1 - o(1))|H| \end{aligned}$$

That is, a would be almost 1!!

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

[12] 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.

[13] 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

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

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

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

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

[18] 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.

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

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

[21] 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

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

[23] Ö. Johansson, personal communication.

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

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

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