On the distributed complexity of computing maximal matchings

Michał Hańćkowiak*
Adam Mickiewicz University
Poznań, Poland

Michał Karoński*
Adam Mickiewicz University
Poznań, Poland

Alessandro Panconesi†
Informatik, HU Berlin
Germany

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 which cannot be solved by deterministic procedures but admit simple randomized solutions. Randomization is also demostrably more powerful in synchronous systems, as shown by the important example of oblivious routing in the hypercube (see, for instance, [13, 19]). 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 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 [15] 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 [20].

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 polylogarithmic time [1, 2, 7, 23]. 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, 16, 21, 23, 24]. 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

*Supported by grant KBN 2 P03A 023 09
†Supported by an Alexander von Humboldt research fellowship..

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 $O(\mathit{polylog}(n))$ 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 [17, 18, 22]. 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 [16]. 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, 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 complexity of our protocol is quite high– $O(\log^7 n)$ rounds– but it should be remembered that even in the EREW-PRAM model the best asymptotic complexity for computing maximal matchings is $O(\log^4 n)$ [9]. Our solution hinges on a distributed procedure 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 assumption on the input network. Other notable exceptions are the so-called ruling forests of [1] and the k-dominating sets of [12] 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 modelled 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 The algorithm: overview and analysis

The starting point of our solution is the NC algorithm for computing maximal matchings, henceforth abbreviated as MM's, due to Israeli and Shiloach [9]. Their algorithm is based on the concept of *spanner* (the terminology is ours). A spanner for a set A of vertices in a given graph G is a subgraph of G of constant degree that contains all vertices of A and only vertices of A. Since a MM in a constant degree graph matches a constant fraction of the vertices, all edges incident on vertices of A can be matched by repeated computations of spanners for A and MM's in the spanner. In this fashion one can first match all edges incident on vertices whose degree is in the interval $[\Delta, \Delta/2]$, then all edges incident on vertices of degree in the interval $[\Delta/2, \Delta/4]$, and so on, until a MM for the whole graph is computed in $O(\log \Delta)$ such stages.

Israeli and Shiloach show how to generate these spanners by means of a splitting procedure that cuts the degree of the vertices almost perfectly in half; if d denotes the old degree of a vertex, the new degree after the split will be between $d/2$ and $(d/2) + 1$. We shall refer to such splitters as *perfect* splitters. Perfect splits can be computed by means of Euler tours, which with a PRAM can be computed efficiently. So, effectively the problem of computing maximal matchings is reduced to that of computing Euler tours.

In implementing the above plan in a distributed setting one immediately faces difficulties of a fundamental character. Namely,

Fact 1 *[14] Neither perfect splitters nor Euler circuits can be computed in $o(n)$ rounds in the synchronous, message-passing model of computation.*

Nevertheless we shall show that the Israeli-Shiloach framework can still be made to work, with some important modifications. In particular, we shall replace the perfect splitters with *approximate splitters.* Our approximate splitter relaxes the splitter of Isreali and Shiloach in three different ways. First, we do not insist that the degree of all vertices be cut in half. Rather, we will be willing to tolerate the loss of a very small fraction of vertices who might lose lots or even all of the edges incident on them. In this way, the final spanner will contain a constant fraction of the initial set of interest, rather than the whole of it. Second, the approximate spanner will not only contain vertices of the initial set under consideration, but also some of their neighbours. These however shall form an independent set, so that computing a MM in the spanner will still match a large fraction of the initial set of vertices. Third, we do not insist that the split be perfect; rather we will require that, for most of the vertices, the new degree be in the range $(1 \pm o(1))d/2$, where d denotes the old degree. The error however will be so small that even after $O(\log n)$ many iterations the distance from the perfect splitting rate will be $(1 \pm o(1))$.

We now describe the protocol. The first step of our solution is to use an idea from [10] to reduce the problem of computing MM's in general graphs to that of computing MM's in bipartite graphs. This is an important step because our approximate splitter cannot deal with odd cycles. The reduction is done by the following recursive procedure, which is also the top level of the overall algorithm.

Procedure MATCH

1. Each vertex u, in parallel, enters a set EVEN or ODD depending on whether the i-th bit of its ID is even or odd, where i is the current level of the recursion.
2. Procedure BIPARTITEMATCH, which computes a MM in the bipartite graph induced by the even and odd vertices, is invoked. Matched edges are added to the solution computed so far.
3. All matched edges and the edges incident on them are removed from the graph. If a vertex remains isolated it removes itself.
4. In parallel, procedure MATCH is invoked recursively on the two vertex disjoint graphs induced, respectively, by the remaining even and odd vertices.

It is apparent that all decision of the algorithm can be made locally, provided the vertices somehow know which edges are matched. The complexity of the algorithm is

$$T(\text{MATCH}, n) = O(\log n \times T(\text{BIPARTITEMATCH}, n)). \tag{1}$$

As the name suggests, the procedure BIPARTITEMATCH computes a maximal matching in the bipartite graph induced by the sets of even and odd vertices. BIPARTITEMATCH deals with high and low degree vertices separately. Here, "high" means of degree higher than any arbitrary constant parameter $t \geq 17$.

The maximum degree of the bipartite graph input of BipartiteMatch is denoted by D. Since the vertices have unique ID's, an upper bound on n, and hence on D is known. For the sake of clarity of exposition we shall assume that the value of D is known globally.

Procedure BipartiteMatch

1. For $i := 0$ to $k = O(\log D)$ do:
   (a) Procedure HighDegreeMatch is invoked to compute a MM in the graph induced by the edges incident on the set $H_i = \{u : D/2^{i+1} \leq d(u) \leq D/2^i\}$. Matched edges are added to the solution computed so far.
   (b) All matched edges and those incident on them are removed from the graph. If a vertex remains isolated it removes itself from the graph.
2. Procedure LowDegreeMatch is invoked to compute a MM in the graph induced by the vertices whose degree is at most t. Matched edges are added to the solution computed so far.
3. All matched edges and those incident on them are removed from the graph. If a vertex remains isolated it removes itself from the graph.

The complexity is

$$\begin{aligned} T(\text{BipartiteMatch}, n) = O(&T(\text{LowDegreeMatch}, t) + \\ &\log D \times T(\text{HighDegreeMatch}, D)). \end{aligned} \tag{2}$$

Notice that all decisions are based on local information. It is clear that if HighDegreeMatch works as advertised, at the end of the for-loop all vertices whose degree is higher than t have disappeared from the graph.

One way to implement procedure LowDegreeMatch would be to resort to a distributed algorithm of Awerbuch et al. for $(t+1)$-vertex colouring graphs of max degree t which runs in $O(t \log n)$ rounds [1]. In our setting however, we can take advantage of the particular structure of the graphs under consideration and use a more direct and efficient approach using $O(t)$ rounds, i.e. constant many rounds. Recall that LowDegreeMatch operates inside a bipartite graph and that the vertices know which side of the bipartition, even or odd, they belong to. Since we shall use this subroutine in different places for graphs of different max degree, we shall endow it with an input parameter t, which in all cases will be known to all vertices executing the procedure.

Procedure LowDegreeMatch(t)

1. Repeat t times:
   (a) Each vertex belonging to the even partition, in parallel, selects any arbitrary edge incident upon itself and proposes it to the other endpoint.
   (b) Each vertex belonging to the odd partition, in parallel, arbitrarily selects one of the proposed edges incident upon itself (if they exist). The selected edge is matched.
   (c) All matched vertices and the edges incident on them are removed from the graph. Isolated vertices remove themselves too.

The complexity is

$$T(\text{LowDegreeMatch}, t) = O(t)$$

because each even vertex will lose at least one incident edge per round.

Next in our top-down description is HIGHDEGREEMATCH. Let $in(A)$ denote the set of edges incident on vertices of a set A and recall that H_i is the set of vertices whose degree is between $D/2^{i+1}$ and $D/2^i$. The task of HIGHDEGREEMATCH is to find a maximal matching in $G[in(H_i)]$, the graph induced by the set of edges incident on vertices in H_i. HIGHDEGREEMATCH invokes another procedure computing a suitable "approximate" spanner in $G[in(H_i)]$.

Definition 2 *An* (α, d)**-spanner** *w.r.t. a graph* G *and a set* $H \subseteq V(G)$ *is a subgraph* G' *of* $G[in(H)]$ *such that:*

- *for every* $v \in V(G')$, $1 \leq \deg_{G'}(v) \leq d$
- $|V(G') \cap H| \geq \alpha |H|$.

Notice that G' might contain vertices not in H but that these form an independent set.

So an (α, d)-spanner is a subgraph of max degree d with no isolated vertices which spans an α-fraction of vertices of H. In the next section we shall show that $d = 16$. The easy proof of the following fact is omitted from this abstract.

Fact 3 *For any constants* α *and* d*, a maximal matching in an* (α, d)*-spanner w.r.t.* G *and* H *matches a constant fraction of vertices of* H.

Thus, all edges incident on vertices of H can be matched in $O(\log |H|)$ rounds as follows.

Procedure HIGHDEGREEMATCH(D)

1. Let $H = \{u : \; D/2 \leq d(u) \leq D\}$.
2. For $k := 1$ to $O(\log n)$ do:
   (a) Invoke the procedure SPANNER, which computes an $(\alpha, 16)$-spanner of $G[in(H)]$.
   (b) Invoke LOWDEGREEMATCH to compute a MM in the $(\alpha, 16)$-spanner.
   (c) The matched edges are added to the current partial solution, and all matched edges, the edges incident on them and isolated vertices are removed from the graph.

Since LOWDEGREEMATCH works in a constant degree graph its complexity is constant. Therefore,

$$T(\text{HIGHDEGREEMATCH}, n) = O(\log n \times T(\text{SPANNER}, n)), \tag{3}$$

where n is the number of vertices of the input set H. As we shall see in the next section, the running time of procedure spanner is $O(\log D \; \log^3 n)$ rounds. Putting together Equations (1) through (3), the overall complexity of the algorithm is

$$T(\text{MATCH}, n) = O(\log^2 D \; \log^5 n) = O(\log^7 n).$$

As the above discussion makes clear, the crux of the matter is how to efficiently compute an (α, d)-spanner, for constant d. The next section explains how this can be achieved.

3 The spanner

In this section we shall show how to compute an (α, d)-spanner in a given graph $G[in(H)]$ induced by the edges incident upon vertices of a set H. 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. Next we partition all long components (paths and cycles) into shorter **segments** , i.e., sub-paths of length at least ℓ but at most $\ell \log n$. Vertices which separate component into segments will be called **border** vertices. Such partitions can be computed distributively in $O(\ell \log n)$ time using the $(\ell, \ell \log n)$-Ruling Sets of [1] and will be referred to as **segment decompositions** (details omitted from this abstract). Roughly speaking one should remove exactly one edge incident on each sibling which in turn would result in slashing the parent's degree perfectly in half (except for an extra edge in case of odd degree). The degree of degree-2 siblings can be cut in half as follows. In each short even cycle we compute a perfect matching (PM) while in paths and long even cycles we might be forced to settle for near PM's (nPM's). Henceforth we will refer to both PM's and nPM's with the latter acronym. The matched edges can be then removed from the graph. For sake of clarity we shall say that the edges are **marked** rather than removed. These nPM's can be computed by resorting to a centralized approach like, say, electing a leader and devolving to it the task. This takes $O(\ell \log n)$ communication rounds since the segment decomposition is made of pieces– segments and short components– of length at most $\ell \log n$. Computing these nPM's in segments and short paths needs a bit of care and will be done as follows; starting from one of the two endpoints, every second edge will be matched. Again, this takes time proportional to the length of the segments, i.e. $O(\ell \log n)$ rounds.

In this fashion we will be able to cut in half the degree of "most" of the siblings so that "very few" parents have "lots" of bad siblings. The trouble makers are border vertices which may be matched or unmatched from each "side" at the same time, and siblings of degree one. The latter present no real problem however, because each parent has at most one of them and, if the segments are long enough, there can't be too many border vertices either. This 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 resulting from a segment decomposition. The value of p, the coefficient of pliability, is $p := \log n$. All parents which are not pliable are called **nasty**. We comment on the need for operating only 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 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 a subroutine APXSPLITTER. SPANNER has input parameter D which is used by the nodes of the network to define the graph inside which an (α, d)-spanner must be computed. This graph is $G[in(H)]$ where

$$H = \{u : D/2 \leq d(u) \leq D\}.$$

Notice that, given D, vertices can decide whether they belong to H locally. A bird's eye view of the algorithm is as follows. Starting from $P_0 := H$, the algorithm computes a segment decomposition of

$G[in(P_0)]$, and nPM's in the resulting segments. 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 segment decomposition of $G[in(P_1)]$ is produced and again nPM's are computed in the 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(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 APXSPLITTER 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 $d_j(u)/p$ bad edges. Let $d_{j+1}(u)$ be the number of unmarked edges incident on u.

Procedure APXSPLITTER acts on P_j, the current set of pliable vertices, by computing nPM's in a segment decomposition of $G[in(P_j)]$ and marking the resulting matched and bad edges.

Procedure APXSPLITTER

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. a segment decomposition is computed in the 2-decomposition;
3. edges incident on border vertices– i.e. vertices separating two segments– enter the set of bad edges;
4. nPM's are computed in the segments;
5. all matched and bad edges are marked.

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 APXSPLITTER. 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.

Fact 4 *For all $v \in P_j$,*

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

where j is any iteration of the for-loop of procedure SPANNER. *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. ♣

The next fact says that most vertices remain pliable from one iteration to the next.

Fact 5 *For all iterations $j = 1, \ldots, k$ of procedure* Spanner*,*

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

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 segments 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 segment 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. ♣

Finally, we shall present the analysis of the procedure Spanner.

Theorem 6 *An invocation of procedure* Spanner *with parameter D computes an $(\alpha, 16)$-spanner w.r.t to G and the set $H := \{v \in V(G) : D/2 \leq deg_v(G) \leq D\}$, where $\alpha > 0$ is some constant.*

Proof: We shall check whether the output graph of our procedure fulfills the two conditions it has to satisfy to be a spanner (see Definition 2). First we show that the degrees of vertices in the output graph belong to the interval $[1, 16]$.

Fix $k \geq 1$ and note that $D/2 \leq d_0(v) \leq D$. By an easy induction, using Fact 4, 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 $q = (1 - 2/\log n)/2$.

Now if we set $k = \log D - c$,

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

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

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

$$|P_k| \geq \alpha\ |H|$$

Applying repeatedly the inequality established in Fact 5, we get

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

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 some constant α and n large enough,

$$|P_k| \geq |H|\Big(1 - \frac{64}{\log n}\Big)^{\log n} \geq e^{-64(1+\frac{64}{\log n})}|H| \geq \alpha|H|.$$

♣

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] S. Chaudhuri and D. Dubhashi, Probabilistic recurrence relations revisited, Theoretical computer Science, to appear.

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

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

[11] 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.

[12] 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

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

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

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

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

[17] 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.

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

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

[20] 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

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

[22] Ö. Johansson, personal communication.

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

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