Efficient Parallel Algorithms for Combinatorial Problems

Oscar Garrido Gómez

Department of Computer Science
Lund University
Sweden

CODEN:LUNFD6/(NFCS-1009)/1-72/(1996)

Department of Computer Science
Lund University
Box 118
S-221 00 Lund
Sweden

Typeset in LaTeX

Pictures where done using xfig and inserted as Postscript[1] figures

[1] Postscript is a registered trademark of Adobe Systems Incorporated

Abstract

This thesis is concerned with the design of efficient parallel algorithms for some optimization graph problems. A graph can be seen as a collection of vertices (V), and a collection of edges (E) joining all or some of the vertices. One is very often interested in finding subsets, either from the set V of vertices or from the set E of edges, which possess some predefined property. A subset S of vertices or edges in a graph G is said to be *maximum* with respect to a property if, among all the subsets of G having this property, S is one having the largest cardinality. Set S is said to be *maximal* with respect to a property, if the set has the property, and it is not a proper subset of another set having this property.

A subset of vertices in a graph, is said to be *independent* if no two of them are adjacent. The problems of finding maximum and maximal independent sets in a graph are well known. One of the more natural generalizations of the notion of independent set is the notion of the so called k-dependent set. For any positive integer k, we call a set Q of vertices a k-dependent set, if each vertex in Q has no more than k neighbours in Q. Thus the notion of independent set is equivalent to the notion of 0-dependent set.

In this thesis we prove that the maximum k-dependent set problem is NP-complete for every fixed integer k even when restricted to planar graphs. We extend some well known approximation heuristics for the maximum independent set problem for planar graphs to include maximum k-dependent sets. We also observe that these problems can be solved in linear time when restricted to graphs with constantly bounded tree-width.

While the problem of finding a maximum k-dependent set in a graph is NP hard, finding a maximal k-dependent set can be easily done in linear time using a greedy algorithm. However, the problem of constructing efficient parallel algorithms for finding a maximal k-dependent set is not trivial. We present the first NC algorithm, i.e., algorithm running in poly-logarithmic time with polynomial number of processors, for this problem.

A subset of the set of edges in a graph such that no two edges in the set are adjacent is called a *matching*, in other words an independent set of edges is a matching. A useful generalization is the notion of f-

matching. Let f be an integer valued function defined over the set of vertices, such that for each vertex v, $0 \leq f(v) \leq \deg(v)$. An f-matching is a subset of edges such that for each vertex v at most $f(v)$ edges of the subset are adjacent to v. Taking $f(v) = 1$ for all vertices v results in an "ordinary" matching.

The problem of finding a maximum matching is sequentially solvable in polynomial time. We don't know whether it is possible to find an NC algorithm for this problem. We show that the maximum f-matching problem is NC reducible to the maximum matching problem. Again, a maximal f-matching can be trivially constructed by a greedy sequential algorithm. Designing parallel algorithms for this problem requires more complicated methods. Here we present the first NC algorithm for this problem running in time $\mathcal{O}(\log^3 n)$ with a linear number of processors and a simple randomized NC algorithm, faster by a logarithmic factor. We also present faster algorithms for this problem for several restricted graph families.

A hypergraph $H = (V, E)$ is a natural generalization of a graph, where an edge in E is an arbitrary subset of the set V. An independent set in H is a subset of V which doesn't include any edge in E. The parallel complexity status of finding a maximal independent set in an arbitrary hypergraph is regarded as a major open problem in parallel complexity theory. We show that this problem admits NC algorithms if the input hypergraph is of so called poly-logarithmic arboricity.

Acknowledgments

This thesis is the result of several years of hard work, and it would never have been finished without the cooperation of many people. I owe thanks to each of them, but there are some I especially wish to mention.

I am greatly indebted to my friend and advisor Prof. Andrzej Lingas for his guidance and friendship, he has always been there when I needed him. Furthermore I wish to thank the people of the Institute of Informatics of Warsaw University, especially Dr. Krzysztof Diks, Prof. Wojciech Rytter and Dr. Stefan Jarominek. Some of the results presented here are the fruit of collaboration with them. I am also grateful to the members of the Algorithm Group at our department, it has been a pleasure to work with them. Particularly I thank Dr. Christos Levcopoulos for carefully reading the manuscript and providing helpful comments. Also thanks go to Anders Dessmark for our interesting and fertile discussions, and to Dr. Bengt Nilsson and Victor Vargas for sharing pleasant moments during the last years.

I am deeply grateful to my wife Norka for her constant encouragement and support and to my children Bianca and Oscar André who make my life enjoyable. This work is therefore dedicated to them.

Finally I would like to thank for the financial support granted by Lund University and the Swedish Research Council for Engineering Sciences (TFR).

To my wife Norka, and my children
Bianca and Oscar André, with love

Contents

Chapter 1

Introduction

Nothing is more important than to see the sources of invention, which are, in my opinion, more interesting than the inventions themselves.

G. W. Leibniz

Many branches of engineering and science rely on graphs for representing a wide variety of objects from electrical circuits, chemical compounds, crystals to genetic processes, sociological structures and economical systems. In many areas of computer science, graphs are used to organize data — to model algorithms as a powerful tool for representing computational concepts. It is therefore important for these applications that efficient algorithms to manipulate graphs are developed.

Since the early days of information processing, one has realized that it is greatly advantageous to have components of a computer to do different things at the same time. Today's most powerful computers contain several processing units sharing jobs submitted for processing. During the last few years *parallel computation* has rapidly become a dominant theme in all areas of computer science and its applications.

The problems whose parallel complexity is the subject of this thesis are natural generalizations of well known graph problems.

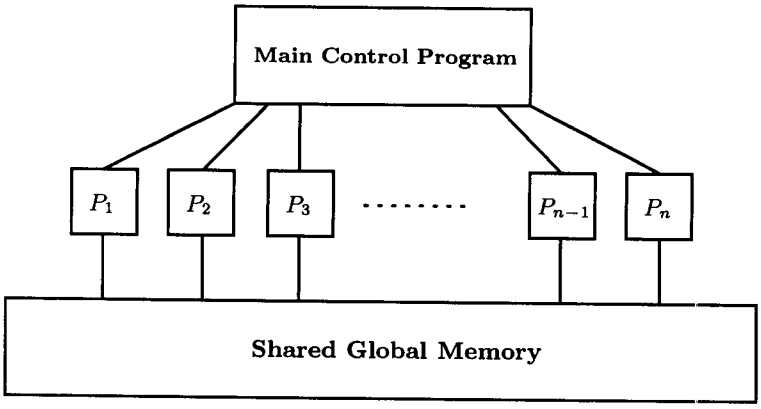


Figure 1.1: The PRAM model of computation.

1.1 A Parallel model of computation

To design and analyze parallel algorithms, we need to introduce the following model of parallel computation.

Definition 1.1.1 *A* **P***arallel* **R***andom* **A***ccess* **M***achine (PRAM) is an idealized model of parallel computation, and can be viewed as the parallel analog of the sequential RAM model. A PRAM consists of several independent sequential processors, each with its own private memory, communicating together through a global memory. In one unit of time, each processor can read one global or local memory location, execute a single RAM operation, and write into a global or local memory location.*

In the PRAM model there is a possibility of read and write conflicts, in which two or more processors try to read or write in the same memory cell concurrently. The differences in the way of handling these conflicts lead to several variants of the model.

- **EREW PRAM** Exclusive **R**ead **E**xclusive **W**rite PRAM
  Only one processor may access any single memory location at the same time.
- **CREW PRAM** Concurrent **R**ead **E**xclusive **W**rite PRAM
  Any memory location can be simultaneously read by many proces-

sors, but can be written to by at most one processor at the same time.

- **CRCW PRAM** Concurrent **R**ead **C**oncurrent **W**rite PRAM
  Any time, many processors can access a given memory location.

The CRCW PRAM requires the application of some rule for writing conflicts, several variations are possible, for example:

- **Common CRCW PRAM**
  All processors writing at the same memory location must write the same value.
- **Arbitrary CRCW PRAM**
  Any of the processors may succeed in writing, and the algorithm should work regardless of which one does.
- **Priority CRCW PRAM**
  There is a linear order among the processors and the minimum numbered processor succeeds in writing.

The above PRAM variants do not differ much in computational power. For example: If an algorithm A on a Common CRCW PRAM takes time T with p processors, it can be simulated on an EREW PRAM in time $T \cdot \mathcal{O}(\log p)$ using p processors [27].

Definition 1.1.2 *Let a PRAM algorithm A solve a problem P of size n in time $T(n)$ using $p(n)$ processors. Then the work $w(n)$ of the algorithm is defined by:*

$$w(n) = p(n) \cdot T(n).$$

Any PRAM algorithm that does $w(n)$ work can be converted to a sequential algorithm that runs in time $\mathcal{O}(w(n))$.

Definition 1.1.3 *An optimal parallel algorithm is an algorithm for which:*

$$w(n) = \mathcal{O}(T_S(n))$$

where $T_S(n)$ is the time expended by the fastest sequential algorithm for the problem.

Definition 1.1.4 *A problem belongs to the class NC introduced in [51] if it can be solved by a PRAM algorithm in poly-logarithmic time using a polynomial number of processors. That is, if $T(n)$ and $p(n)$ respectively stand for the time and the number of processors used by the algorithm then*

$$T(n) = \mathcal{O}(\log^k n)$$
$$p(n) = \mathcal{O}(n^\ell)$$

for some integer constants k and ℓ.

Definition 1.1.5 *More refined notions of parallel complexity are obtained by defining the following family of important subclasses of NC: NC^k is the subset of NC in which the parallel time is $\mathcal{O}(\log^k n)$. Here we allow any degree in the polynomial bounding the number of processors.*

Definition 1.1.6 *Analogously, the randomized NC class (RNC) is the class of problems computable by a randomized PRAM algorithm in poly-logarithmic expected time using a polynomial number of processors.*

Usually it is much easier to design RNC algorithms than NC algorithms. If we succeeded in showing a problem to be in NC, the next step is to design an optimal NC-algorithm for the problem.

There are many problems admitting polynomial-time sequential algorithms which do not seem to admit fast parallel algorithms, e.g., the so-called P-complete problems.

1.2 Graph terminology

We shall use in particular the following graph conventions and notations. By a graph we mean a simple graph, i.e., a graph without self-loops and multi-edges. Let $G = (V, E)$ be a graph. We denote the number of vertices of a graph by n ($|V| = n$), and the number of edges by m, ($|E| = m$). For any set $S \subseteq V$, we define the *neighbourhood* of S in G as:

$$N_G(S) = \{w \in V \mid \text{ there exists } u \in S \text{ such that } (u, w) \in E\}.$$

In the same way, the set of *neighbours* of a node v in the graph G is the set $N_G(v) = N_G(\{v\})$.

When it is clear in the context which graph we are referring to, we use the notation $N(S)$ and $N(v)$, instead of $N_G(S)$ and $N_G(v)$.
The number of neighbours of a vertex v in a graph G is called the *degree* of v in G and denoted as $\deg_G(v)$. The maximum degree in a graph is called its *valence*. In a similar manner, for a given subgraph H of G, and for any vertex v of G, we define the *degree* of v in H as the number of neighbours of v which are vertices of the subgraph H, and denote it as $\deg_H(v)$. Note that in the last definition the vertex v doesn't need to be in H.
If $S \subseteq V$ then $\gamma(S)$ will denote the *subgraph induced by* S. This subgraph has vertex set S, and its edge set $E_{\gamma(S)}$ consists of these edges in E that are incident only to vertices in S.

The arboricity $\Upsilon(G)$ of a graph G is the minimum number of forests the edges of G can be partitioned into. For example, graphs of bounded genus and partial k-trees have constant arboricity.

1.3 Optimization graph problems

One is often interested in finding subsets, either from the set V of vertices or from the set E of edges, which possess some predefined property. A subset S of vertices or edges in a graph G is said to be *maximum* with respect to a property if, among all the subsets of G having this property, S is one having the largest cardinality. Set S is said to be *maximal* with respect to a property, if the set has the property, and it is not a proper subset of another set having the property.

1.3.1 Independent set problems

A subset of vertices in a graph, is said to be *independent* if no two of them are adjacent, i.e, no two vertices are joined by an edge.

Maximum independent sets

The problem of determining whether a given graph contains an independent set of prescribed size is NP-complete [20]. Even if we restrict our attention to cubic planar graphs the problem remains NP-complete [20]. This implies that there is no realistic hope to design polynomial time algorithms to exactly solve the problem. As for many optimization

problems the task of finding a good approximate solution for maximum independent set is often as difficult as that for finding the optimum solution [6]. Therefore efficient approximation algorithms have been designed for the problem restricted to special graph classes. These kind of algorithms are evaluated by the *worse-case ratio*: the smallest ratio of the size of the approximated solution over the size of the maximum solution.

For planar graphs Lipton and Tarjan using their well known *Planar separator theorem* [43], designed a $\mathcal{O}(n \log n)$ time approximation algorithm with a worse-case ratio of $1 - \mathcal{O}(\frac{1}{\sqrt{\log \log n}})$ asymptotically approaching 1 as $n \to \infty$ [43]. Another approach was used by Baker who found a linear time approximation algorithm running in time $\mathcal{O}(8^k kn)$ for any positive integer k with the worse-case ratio of $\frac{k}{k+1}$ [4].

The maximum independent set problem can be expressed in linear extended monadic second order logic. Consequently it can be solved in linear time if the input graph is of bounded tree-width and is provided with its tree-decomposition [1].

Maximal independent sets

While the problem of constructing a maximum cardinality independent set is NP-hard [20], the problem of constructing a maximal independent set (MIS for short) can be trivially solved in linear time (See Algorithm 1.1). However, the problem of constructing an efficient NC algorithm for MIS is non-trivial. For a period of time it was believed that the maximal independent set problem was one of the problems for which there is no NC algorithm. Algorithm 1.1 clearly runs in linear time. The maximal independent set output I obtained by this algorithm is called *the lexicographically first maximal independent set* (LFMIS for short). Cook [12] showed that the LFMIS problem is NC-complete for P. Karp and Widgerson [39] were the first who proved that the problem is in NC^4. They presented a parallel randomized algorithm with expected running time $\mathcal{O}(\log^4 n)$ using $\mathcal{O}(n^2)$ processors on a EREW PRAM, and a deterministic algorithm with running time $\mathcal{O}(\log^4 n)$ using $\mathcal{O}(\frac{n^3}{\log^3 n})$ processors also on a EREW PRAM. Presently, the most efficient deterministic NC algorithm is due to Goldberg and Spencer [29]. It runs in time $\mathcal{O}(\log^3 n)$ and uses an EREW PRAM with a linear number of

Algorithm *Sequential-Maximal-Independent-Set*(G)
input : A graph $G = (V, E)$
output : A maximal independent set I on G.
method :

```
I ← ∅
for v ∈ V do
    if v ∉ N(I) then
        I ← I ∪ {v};
    endif
endfor
output I;
```

end *Sequential-Maximal-Independent-Set*

ALGORITHM 1.1

processors. Luby has constructed a randomized parallel algorithm for MIS that runs in time $\mathcal{O}(\log^2 n)$ and uses a EREW PRAM with $\mathcal{O}(nm^2)$ processors [45].

Algorithm *Parallel-Maximal-Independent-Set*(G)
input : A graph $G = (V, E)$
output : A maximal independent set I on G.
method :

```
I ← ∅
A ← V
while A ≠ ∅ do
        C ← FINDSET(A);
        I ← I ∪ C;
        A ← A \ (C ∪ N(C))
endwhile
output I;
```

end *Parallel-Maximal-Independent-Set*

ALGORITHM 1.2

Both Karp's, and Goldberg-Spencer's algorithms use the same top level description, (Algorithm 1.2), differing only in the procedure FIND-SET.

It is easy to prove that an algorithm with the top structure of Algorithm 1.2 has a poly-logarithmic running time if every call to the function FINDSET produces an independent set C such that $|C \cup N(C)| = \Omega(\frac{|A|}{\log^s |A|})$ for some fixed $s \geq 0$.

Luby [45] derived (with the same top level description) a Monte Carlo algorithm which can be implemented on a EREW PRAM with $\mathcal{O}(nm^2)$ processors with running time $\mathcal{O}(\log^2 n)$.

Gazit and Miller used their planar separator algorithm to derive an algorithm which for planar graphs constructs an independent set of size within $1 - \mathcal{O}(\sqrt{\frac{1}{\log \log n}})$ from the maximum and runs in time $\mathcal{O}(\log(n) \cdot \log\log(n))$ on a probabilistic PRAM with $n^{1+\epsilon}$ processors for arbitrary small given ϵ.

1.3.2 Maximal independent sets in hypergraphs

A hypergraph $H = (V, E)$ is a natural generalization of a graph. Each edge of E in H is a non-empty subset of V. For a vertex v in V, the degree $\deg(v)$ of v in H is the number of edges in E it belongs to. The maximum degree of a vertex in V is called the valence of H, and the maximum cardinality of an edge in E is called the dimension of H. A hypergraph of dimension two is simply an undirected graph (if singletons are neglected). An independent set of H is a subset of V which doesn't include any edge in E. A maximal independent set (MIS, for short) of H is an independent set which is not a subset of any other independent set of H.

A MIS of a hypergraph can be trivially computed in polynomial time by a greedy method. The parallel complexity status of finding a MIS of an arbitrary hypergraph is regarded as a major open problem in parallel complexity theory (See Section 4.2 for details).

1.3.3 Matching problems

A subset of the set of edges in a graph such that no two edges in the set are adjacent is called a *matching*. In other words, a matching is a subset of the set of edges of G in one-to-one correspondence with an independent set in the edge graph induced by G. In the edge graph, the vertices correspond to the edges of G and two such vertices are adjacent if and only if the corresponding edges of G are incident in G.

Maximum matchings

A *maximum matching* is a matching of maximum cardinality, that is, a matching M such that for any matching M', $|M| \geq |M'|$ holds.

Contrary to the maximum independent set problem which is known to be NP-complete, the maximum matching problem admits polynomial time solutions. The best known sequential algorithm for the latter problem is due to Micali and Vazirani [48] and runs in time $\mathcal{O}(\sqrt{n} \cdot m)$.

Finding a maximum matching in a graph is a fundamental problem in combinatorial optimization. It is a major open problem whether a maximum matching can be constructed by an NC algorithm. Achieving simultaneously a poly-logarithmic-time and a polynomial number of processors is possible for this problem if random bits are used. Randomized NC algorithms have been given for constructing maximum matching [19, 35, 37, 49]. They also yield randomized NC algorithms for several other problems not known to be in NC, e.g., depth first search, maximum network flow with polynomially bounded edge capacities, maximum weight matching with polynomially bounded edge weights [36, 37].

Maximal matchings

A matching is *maximal* if it is not a proper subset of any other matching.

While the best known sequential algorithm for constructing a maximum cardinality matching runs in time $\mathcal{O}(\sqrt{n} \cdot m)$, the problem of finding a maximal matching (MM for short) admits trivial linear-time sequential solutions. Also, the known NC algorithmic solutions to MIS can be specialized to MM. However, as the edge graph may have a quadratic number of edges with respect to the size of the input graph, more efficient NC algorithms for MM can be derived directly. Presently, the most efficient deterministic algorithm for MM is due to Israeli and Shiloach [34]. It can be implemented in time $\mathcal{O}(\log^3 n)$ on a CRCW PRAM with a linear number of processors, and consequently in time $\mathcal{O}(\log^4 n)$ on an EREW PRAM with a linear number of processors [36]. The RNC algorithm for the same problem due to Itai and Israeli [33] works in time $\mathcal{O}(\log n)$ on an CRCW PRAM with $\mathcal{O}(n + m)$ processors.

Chapter 2

k-dependent Sets

I ain't get nobody, that I can depend on,
No tengo a nadie...

Carlos Santana

Let k be a non-negative integer. A *k-dependent set* in a graph G is a subset of the set of vertices of G such that no vertex in the subset is adjacent to more than k vertices of the subset. This subset induces a subgraph of G of maximum degree bounded by k.

Note that a 0-dependent set in G simply means an independent set of vertices on G. Further, 1-dependent set is in general a set of independent vertices and edges whereas a 2-dependent set is a set of independent paths, possibly degenerated, such that no two non-consecutive vertices on any of these paths are adjacent. These paths are called *permissible* [7].

2.1 Maximum k-dependent set problem

Maximum cardinality k-dependent sets, for $k = 2$, have applications in information dissemination in hypercubes with large number of faulty processors. The cardinality of a largest 2-dependent set in a network is an upper bound on the length of maximum permissible path in the network which in turn bounds the time of dissemination of information in the network when the majority of the processors are faulty [7].

We shall term the problem of finding a maximum cardinality k-dependent set in a graph as the *maximum k-dependent set problem.* The latter is a natural generalization of the maximum independent set problem resembling such problems as the NP-complete degree constrained subgraph problem and the f-matching problem solvable in polynomial time (see [20]). We shall also term the union of the maximum k-dependent set problems, as the maximum multi-dependent set problem. An instance of the above problem consists of a graph and a non-negative integer k, and the solution is a maximum k-dependent set for the graph. As the maximum independent set problem trivially reduces to the maximum multi-dependent set problem, the latter is clearly NP-complete.

2.1.1 NP-completeness

The decision version of the maximum k-dependent set is as follows: For a non-negative integer ℓ and a graph $G = (V, E)$, decide whether there is a k-dependent set of cardinality not less than ℓ in G.

Theorem 2.1.1 *For any non-negative integer k, the decision version of the maximum k-dependent set is NP-complete.*

Proof: The maximum k-dependent set problem is clearly in NP since we need only to guess a subset Q of V of size ℓ and check in polynomial time that for each vertex $v \in Q$, $\deg_{\gamma(Q)}(v) \leq k$.

To prove that the maximum k-dependent set problem is NP-complete, it suffices to show that the maximum independent set problem is many-one polynomial-time reducible to the maximum k-dependent set problem, for any positive integer k.

Let G be a graph on n vertices. Let H be the graph constructed in the following way:
Copy the original graph G. Now hang k pendants on all the original vertices.
By definition all pendant vertices form an independent set S. By the construction this independent set is maximal. We are going to show that there exists a maximum k-dependent set S in H which contains all pendant vertices.
Let S' be a maximum k-dependent set in H which does not contain all pendant vertices. Let w be a pendant vertex in H which is not in S', that means that the neighbour v of w is in S' and that v has k-neighbours in

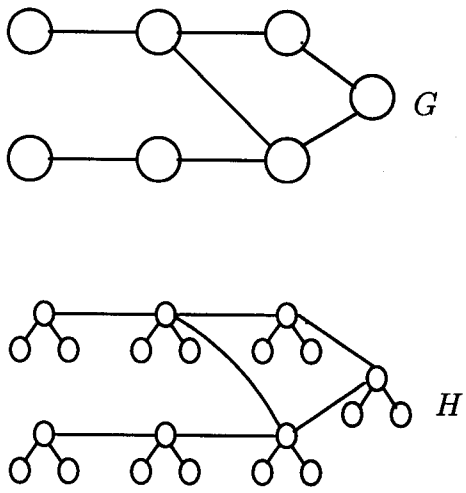


Figure 2.1: Example of the reduction of the maximum independent set problem to the maximum k-dependent set problem for $k = 2$.

S', which implies that at least one not pendant neighbour u of v is in S'. We can now construct another k-dependent set S by deleting u and adding w and all pendant neighbours of u into S'. Clearly $|S| \geq |S'|$. Repeating this procedure for all pendant vertices which are not in S' we obtain a maximum k-dependent set S in H which contains all pendant vertices.

Note that $|S| \leq$ the number of pendants plus the cardinality of a maximum independent set of G, also the reverse inequality holds, since the union of the set of pendants and a maximum independent set in G is k-dependent. Suppose now that we can construct a maximum k-dependent set in polynomial time. Then to construct a maximum independent set of G, we just construct the graph H, compute a maximum k-dependent set in H, construct the maximum k-dependent set which contains all pendant vertices, and clearly the non-pendant vertices of the maximum k-dependent set form a maximum independent set. Observe that the reduction used here preserves planarity. □

2.1.2 Fast algorithms for trees

The NP-completeness of the decision version of the maximum k-dependent set does not exclude the possibility of the existence of efficient algorithms for constructing a maximum cardinality k-dependent set when the input graph ranges over a more restricted graph family. Below we give a simple linear-time algorithm (Algorithm 2.1) for the maximum multi-dependent set problem in trees.

The idea (Algorithm 2.1) is to root the input tree T, and process the vertices v of the tree, whose all sons have been already processed, as follows: augment Q by v if Q remains k-dependent (note that in particular all leaves become elements of Q). Assume inductively that there is a maximum cardinality k-dependent set S in T that includes all vertices from the current Q and excludes all other vertices processed before the vertex v. If Algorithm 2.1 inserts v into Q and S does not include v then the father of v is in S. Now by deleting the father of v from S and inserting v instead we obtain another maximum cardinality k-dependent set S' which includes the current Q after inserting v into Q. In this way the correctness of the algorithm follows.

Theorem 2.1.2 *A maximum cardinality k-dependent set in a tree on n vertices is constructed by Algorithm 2.1 in time bounded by cn where c is a positive constant independent of k. In other words, Algorithm 2.1 solves the maximum multi-dependent set problem in linear time.*

Proof: The correctness of Algorithm 2.1 follows from the argumentation preceding its definition. Its linear time performance is a consequence of the fact that each vertex v in T is inserted on L only once and its processing when picked up from L takes a constant number of operations independent of k. □

Using Algorithm 2.1 we can find tight lower bounds on the maximum cardinality of k-dependent sets, as it is stated in the following theorem:

Theorem 2.1.3 *A maximum cardinality k-dependent set in any tree with n vertices contains at least $\lceil \frac{k+1}{k+2} n \rceil$ vertices, and this bound is tight for any integers k and n, $k \geq 0$ and $n \geq 1$.*

Proof: In Algorithm 2.1 for each vertex v not in Q the following holds: either v has at least $k+1$ sons in Q or v has some son v' such that

```
Algorithm  Maximum-Cardinality-k-Dependent-Set(T)
input :    A non-negative integer k and a tree T.
output :   A maximum cardinality k-dependent set Q in T.
method :
           root T
           Q ← ∅
           L ← empty list
           for all vertices v ∈ T do
               sons-in-Q(v) ← 0
               waiting-sons(v) ← the number of sons of v in T
               candidate(v) ← true
               if v is a leaf then
                  insert v on L
               endif
           endfor
           while L is not empty do
                  pop a vertex v from L
                  if v is the root of T then
                     if candidate(v) and (sons-in-Q(v) ≤ k) then
                        insert v into Q
                     endif
                     output Q
                     exit
                  endif
                  f ← father of v in T
                  if candidate(v) and (sons-in-Q(v) ≤ k) then
                     insert v into Q
                     sons-in-Q(f) ← sons-in-Q(f) + 1
                  endif
                  if candidate(v) and (sons-in-Q(v) = k) then
                     candidate(v) ← false
                  endif
                  waiting-sons(f) ← waiting-sons(f) −1
                  if waiting-sons(v) = 0 then
                     insert f into L
                  endif
           endwhile
end  Maximum-Cardinality-k-Dependent-Set
```

ALGORITHM 2.1

v' is in Q and v' has exactly k sons in Q. Thus for each vertex not in Q there are at least $k+1$ distinct vertices in Q, so the lower bound follows. To see, on the other hand, that for any integers $k \geq 0$ and $n \geq 1$ there exists a tree such that any k-dependent set of the tree has at most $\lceil \frac{k+1}{k+2} n \rceil$ vertices, consider $\lfloor \frac{n}{k+2} \rfloor$ copies of a star graph, each consisting of one vertex of degree $k+1$ and $k+1$ vertices of degree one. Let T be some tree obtained by connecting those copies through edges and adding some additional vertices if necessary, so that the total number of vertices becomes n. We observe that for each copy of the star at most $k+1$ of its vertices can be in the same k-dependent set. Hence no k-dependent set of T has more than $\lceil \frac{k+1}{k+2} n \rceil$ vertices. □

In order to obtain a work-optimal logarithmic time parallel implementation of Algorithm 2.1 in a PRAM model we can use the parallel expression evaluation algorithm (see [10, 27]). In this way we obtain the following result.

Theorem 2.1.4 *A maximum cardinality k-dependent set in a tree on n vertices can be constructed in time $\mathcal{O}(\log n)$ using an EREW PRAM with $\mathcal{O}(\frac{n}{\log n})$ processors.*

For a fixed k, the maximum k-dependent problem can be relatively easily encoded by a formula of the extended monadic second order logic which by the general fact proved in ([1], Section 5) yields the following theorem:

Theorem 2.1.5 *Given a non-negative integer k, the problem of finding a maximum k-dependent set in a graph of tree-width bounded by a constant is solvable in linear time if the graph is given together with its tree decomposition.*

The drawback of the above solution to the maximum k-dependent set problem for graphs of bounded tree width is a high constant of proportionality which is a monotone function of the bound on the tree width and k plus the requirement of having a constant width tree decomposition of the graph. Of course, in case of trees, we have the decomposition for free, and consequently the above theorem also implies a linear time-bound on the construction of a maximum k-dependent set for a tree. However, the bound depends on k and therefore it is subsumed by that given in Theorem 2.1.2 yielding solution to the maximum multi-dependent set problem.

2.1.3 Approximation for planar graphs

The decision version of the maximum independent set problem is known to remain NP-complete when restricted to planar graphs [20]. As the reduction given in the proof of Theorem 2.1.1 preserve planarity we conclude:

Theorem 2.1.6 *The maximum k-dependent set problem remains NP-complete even when the input graph is planar.*

Here we only observe that the known approximation heuristics for constructing a maximum independent set for planar graphs (see [4, 43]) can be easily generalized to include maximum k-dependent set.

Theorem 2.1.7 *Let k be a non-negative integer, and let G be a planar graph on n vertices. A k-dependent set in G of size within $1 - \mathcal{O}(\frac{1}{\sqrt{(\log \log n)}})$ from the maximum can be constructed in time $\mathcal{O}(\frac{n}{\log n})$.*

Proof: Adopt the approximation method for maximum independent set due to Lipton and Tarjan [43] that recursively splits the input graph into pieces of size $\mathcal{O}(\log \log n)$ using a version of the planar separator theorem. For each of the pieces apply an exhaustive search algorithm in order to find a maximum k-dependent set within it. The union of the optimal solutions for the pieces is the sought approximation. □

Theorem 2.1.8 *Let k be a non-negative integer, let ℓ be a positive integer, and let G be a planar graph on n vertices. A k-dependent set in G of size within $\frac{\ell}{\ell+1}$ from the maximum can be constructed in linear time.*

Proof: Adopt the approximation method for maximum independent set due to Baker [4] that splits G into $\ell+1$ split graphs whose connected components are ℓ-outerplanar graphs. For each of the split graphs find a maximum cardinality k-dependent set. As an ℓ-outerplanar graph is a graph of $\mathcal{O}(\ell)$ tree width, the latter can be done in linear time by Theorem 2.1.5. Choose the largest of the above sets as the approximation. □

We can also derive poly-logarithmic time implementations of the approximation algorithms for maximum k-dependent set in planar graphs

which use polynomial numbers of processors. For instance, to obtain a parallel analogue of Theorem 2.1.7, we could use the poly-logarithmic time algorithm for simple cycle separator in planar triangulated graphs [26] due to Gazit and Miller (we would add dummy edges to triangulate the input graph).The algorithm uses random bits and $n^{1+\epsilon}$ processors for arbitrary small given ϵ. This leads to the following proposition.

Theorem 2.1.9 *Let k be a non-negative integer, and let G be a planar graph on n vertices. A k-dependent set in G of size within $1 - \mathcal{O}(\sqrt{\frac{1}{\log\log n}})$ from the maximum can be constructed in poly-logarithmic time using a probabilistic PRAM with $n^{1+\epsilon}$ processors for arbitrary small given ϵ.*

2.2 Maximum f-dependent set problem

Let $f(V) \mapsto \mathcal{Z}^+$ be a non-negative function defined on the vertex set V of a graph G. A f-dependent set in G is a subset $\mathcal{F}$ of the vertices of G such that no vertex $v \in \mathcal{F}$ is adjacent to more than $f(v)$ vertices in $\mathcal{F}$. Clearly the notion of f-dependent set is a natural generalization of of the notion of k-dependent set.

Theorem 2.2.1 *The decision version of the maximum f-dependent set is NP-complete for planar graphs.*

Proof: Generalize the proof of Theorem 2.1.6 by constructing H in the following way: for each v in G, hang $f(v)$ pendants on the copy of the vertex v. □

2.3 Maximal k-dependent set problem

We can naturally generalize the notion of maximal independent sets to include maximal k-dependent sets. Again, the problem of finding a maximal k-dependent set can be solved by a trivial greedy algorithm in linear time. At this point it is natural to ask whether this generalized problem admits NC algorithms.
In this chapter we give an affirmative answer to the above question. We

present the first NC algorithm for constructing a maximal k-dependent set in a graph G on n vertices [16]. The algorithm runs in time $\mathcal{O}(\log^3 n)$ using an EREW PRAM with $\mathcal{O}(n^2)$ processors. We also observe that if G has bounded valence then it can be modified to run in time $\mathcal{O}(\log^* n)$ on an EREW PRAM with a linear number of processors. The algorithm can be also easily generalized to find a maximal f-dependent set in G in time $\mathcal{O}((\log^3 n) \cdot (\max_{v \in V} f(v)^2))$ using an EREW PRAM with $\mathcal{O}(n^2)$ processors.

2.3.1 A NC algorithm for maximal k-dependent set

Our parallel algorithm for maximal k-dependent ($k > 0$) set can be seen as an NC Turing-like reduction to the problem of constructing a maximal 0-dependent (independent) set (MIS). Recall that the best known parallel algorithm for MIS runs in time $\mathcal{O}(\log^3(n))$ using $\mathcal{O}(n+m)$ EREW processors [29]. We shall analyze our algorithm also in terms of the EREW shared memory model, where simultaneous reads and writes into the same memory locations are not permitted.

Algorithm *MDS*(G)
input : A graph $G = (V, E)$.
output : A maximal k-dependent set Q for G.
method :

$Q \leftarrow$ *Maximal-Independent-Set*(G);
$R \leftarrow V \setminus Q$;
$B \leftarrow \{v \in R \mid \deg_{\gamma(Q)}(v) \leq k\}$;
while $B \neq \emptyset$ **do**
  $H \leftarrow$ the graph whose set of vertices is B
    such that (v, w) is an edge of H iff
    v and w have a common neighbour in Q
    or (v, w) is also an edge in G;
  $M \leftarrow$ *Maximal-Independent-Set*(H);
  $Q \leftarrow Q \cup M$;
  $S \leftarrow \{v \in Q \mid \deg_{\gamma(Q)}(v) = k\}$;
  $R \leftarrow R \setminus (M \cup N_{\gamma(R)}(S))$;
  $B \leftarrow \{v \in R \mid \deg_{\gamma(Q)}(v) \leq k\}$;
endwhile
output Q;
end *MDS*

ALGORITHM 2.2

Lemma 2.3.1 *Algorithm 2.2 is partially correct, i.e., if it stops then the set Q to output is a maximal k-dependent set in G.*

Proof: It is sufficient to observe that the augmentation of Q by M is correct since M is in particular independent in G, no two vertices in M share a common neighbour in Q and no vertex in M is a neighbour of a vertex in Q that has already k neighbours in Q. □

Lemma 2.3.2 *The block under the while statement is iterated $\mathcal{O}(k^2)$ times.*

Proof: For a vertex $v \in G$, let $\mathrm{cap}(v) = \min(\deg_G(v), k) - |N_{\gamma(Q)}(v)|$ at a given stage of performance of Algorithm 2.2. Consider a vertex $v \in B$ at the beginning of the i:th iteration of the block. Next, let $g(v) = \sum_{w \in N_{\gamma(Q)}(v)} \mathrm{cap}(w)$. Note that $g(v)$ is always bounded by k^2 from above. Also, if v disappears from B in some of the next iterations then it never can reappear in B. On the other hand, after each iteration, if v remains in B then there exists a vertex w newly inserted into Q that either shares a neighbour in Q with v or it is itself a neighbour of v in G. In the first case, $g(v)$ decreases at least by 1. In the second case $g(v)$ increases by $\mathrm{cap}(w)$, i.e., at most by k. However, the second case can occur at most k times since $\mathrm{cap}(v)$ cannot be negative if v is to stay in B. Since $\mathrm{cap}(w)$ for each neighbour w of v in Q has to be positive in order to keep v in B, we conclude that after the $k^2 + k$ iterations v has to disappear from B.
Suppose that v is not a member of the original set B. It means that v has no neighbour in the original set Q. Therefore, it could be added to the original set Q preserving its independence property which would contradict the maximality of the set. We obtain a contradiction. Thus, we can conclude that all vertices that appear in the sets B are members of the original set B. Combining this conclusion with the shown fact that no member of B can survive more than the $k^2 + k$ iterations we obtain the thesis of the lemma. □

Combining the two above lemmas, we obtain the correctness of Algorithm 2.2.

Theorem 2.3.1 *Algorithm 2.2 is correct.*

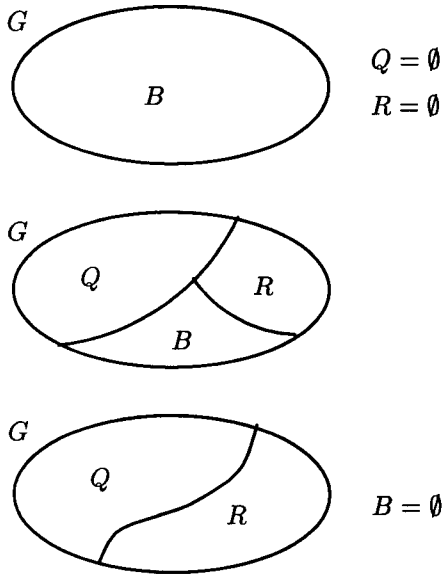


Figure 2.2: The idea of how to construct a maximal k-dependent set. B is the set of vertices which we still do not know if are in the k-dependent set, Q is a k-dependent set and R is the set of vertices we know to be outside our k-dependent set Q.

Lemma 2.3.3 *Suppose that a maximal independent set in a graph on n vertices can be found in time $T(n)$ using an EREW PRAM with $P(n)$ processors. Algorithm 2.2 can be implemented in time $\mathcal{O}(\log n + T(n))$ using a PRAM with $\mathcal{O}(n^2 + P(n))$ processors.*

Proof: A maximal independent set in G as well as a maximal independent set in the auxiliary graph H can be found in time $T(n)$ using $P(n)$ processors. By Lemma 2.3.2, we can replace the while statement by a "for" loop with the number of iterations $\mathcal{O}(k^2)$, in this way avoiding the test for emptiness of B.

All other instructions except for the construction of the auxiliary graph H can be easily implemented in time $\mathcal{O}(\log n)$ using an EREW PRAM with $\mathcal{O}(n^2)$ processors. For instance, filtering B out of R can be implemented using the sorting algorithm due to Cole [9] running in a logarithmic time on an EREW PRAM with a linear number of processors

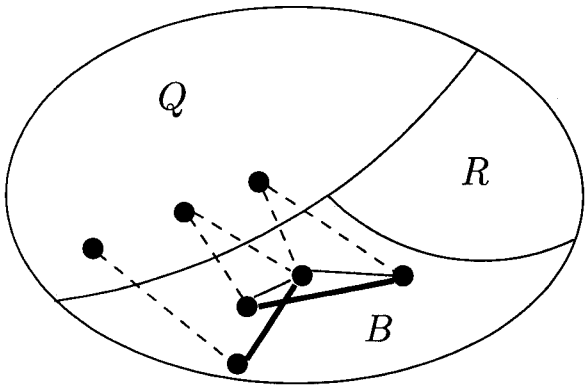


Figure 2.3: Constructing the graph H (solid lines) whose set of vertices is B and the edges are the edges in G (bold lines) and the edges (u, v) if u and v share a common neighbour in Q.

($\mathcal{O}(n^2)$ processors in our application).

We can implement the set of operations in constant time by representing all the involved sets B, Q, R, S with n element vectors, each of them with 1 on the i:th position if and only if the i:th vertex in G is currently in the set. The construction of the auxiliary graph H, in particular finding all pairs (v, w) such that v and w have a common neighbour in Q, seems to be more costly. It immediately reduces to the following problem:

Given a bipartite graph $F = (V_1, V_2, E)$, where the degree of each vertex in V_2 is bounded by k, construct the graph $F' = (V_2, E')$ such that (v, w) is in E' if and only if v and w have a common neighbour in V_1. Suppose that $V_2 = \{1, 2, \ldots, s\}$. Construct a matrix of size $s \times s$ such that $W(i, j)$, $1 \leq i \leq j \leq s$, is set to the list of neighbours of i in F (i.e., in V_1). Note that such a list can have at most k elements. For this reason, the matrix can be constructed in time $\mathcal{O}(\log s)$ using an EREW PRAM with $\mathcal{O}(s^2)$ processors.

Now the construction of the graph F' becomes easy. For each pair i, j where $1 \leq i \leq j \leq s$, we check whether the lists $W(i, j)$ and $W(j, i)$ have at least one element in common. If so we augment E' by (i, j), i.e., we set to one the corresponding entry of the adjacency matrix of F'. Note that comparing two such lists takes time $\mathcal{O}(k)$.

We conclude that F' can be constructed in time $\mathcal{O}(\log s)$ using an EREW PRAM with $\mathcal{O}(s^2)$ processors. Consequently, the auxiliary graph H can be constructed in time $\mathcal{O}(\log n)$ using an EREW PRAM with $\mathcal{O}(n^2)$ processors. As by Lemma 2.3.2, all instructions within the loop are executed only $\mathcal{O}(k^2)$ times, we conclude that Algorithm 2.2 can be implemented in time $\mathcal{O}(\log n + T(n))$ using $\mathcal{O}(n^2 + P(n))$ processors. □

Theorem 2.3.2 *Let k be a non-negative integer. A maximal k-dependent set in a graph on n vertices can be computed in time $\mathcal{O}(\log^3 n)$ using an EREW PRAM with $\mathcal{O}(n^2)$ processors.*

Proof: As a maximal independent set in a graph on n vertices can be computed in time $\mathcal{O}(\log^3 n)$ using an EREW PRAM with $\mathcal{O}(n^2)$ processors we obtain the thesis by Lemma 2.3.3. □

Corollary 2.3.4 *The problem of constructing a maximal k-dependent set is in NC.*

2.3.2 A maximal k-dependent set algorithm for bounded degree graphs

A maximal independent set in a graph of bounded valence on n vertices can be computed in time $\mathcal{O}(\log^* n)$ using an EREW PRAM with $\mathcal{O}(n)$ processors [30]. We can use this fact to speed-up Algorithm 2.2 in the case of graphs of bounded valence [22]. Here we present a $\mathcal{O}(\log^* n)$ time algorithm for the maximal k-dependent set problem which uses colouring.

Given a graph $G = (V, E)$ with maximum degree $= \Delta$, it is obvious that for any number $k \geq \Delta$ the maximum (and of course the maximal) k-dependent set in G is the set of vertices V itself. So in this section we assume that $k < \Delta$.

The fact that the Algorithm 2.3 computes a maximal k-dependent set is obvious. The part of the algorithm which requires a detailed explanation is the selection of sets W_i. We remark that no vertex $u \in V_i$ with more than k neighbours in Q or with a neighbour $v \in Q$ with exactly k neighbours in Q can belong to W_i.

Let $U_i \subseteq V_i$ be the set of vertices in V_i with at most k neighbours in Q, such that no one of their neighbours in Q have their degree in $\gamma(Q)$

Algorithm *Maximal-k-dependent-set*(G)
input : A graph $G = (V, E)$ with bounded degree Δ.
output : A maximal k-dependent set Q.
method :

```
Colour vertices of G using Δ + 1 colours;
Let V_1, V_2, ..., V_{Δ+1} be the partition
of V into colour classes;
Q ← V_1;
for i ← 2 to Δ + 1 do
    W_i ← a subset of V_i such that
          Q ∪ W_i is a maximal k-dependent
          set in γ(V_1 ∪ ... ∪ V_i);
    Q ← Q ∪ W_i;
endfor
output Q;
```

end *Maximal-k-dependent-set*

ALGORITHM 2.3

equal to k.
Now we must select $W_i \subseteq U_i$ in such a way that $Q \cup W_i$ is a maximal k-dependent set in $\gamma(V_1 \cup \ldots \cup V_i)$. Q is already a maximal k-dependent set in $\gamma(V_1 \cup \ldots \cup V_{i-1})$, so we must assure that W_i does not violate the k-dependency. For this purpose we create a graph $G_i = (U_i, E_i)$ with the edge set defined as:

$$(u, v) \in E_i \Longleftrightarrow u \text{ and } v \text{ have a common neighbour in } Q.$$

Adjacent vertices in G_i cannot be inserted simultaneously in W_i, because it could violate the k-dependency of $Q \cup W_i$. Since G_i is a bounded degree graph with maximum degree Δ_i ($\Delta_i \leq \Delta^2$), we can rapidly colour G_i with $\Delta_i + 1$ colours. Call $U_{i,1}, U_{i,2}, \ldots, U_{i,\Delta_i+1}$ the colour partition of U_i. Now we can insert into W_i the elements of $U_{i,j}$ checking for every element that it does not violate the k-dependency. We can do it in parallel because each set $U_{i,j}$ is independent. Hence by the above discus ions we obtain:

Theorem 2.3.3 *Let k be a non-negative integer. A maximal k-dependent set in a graph of bounded valence on n vertices can be computed in time $\mathcal{O}(\log^* n)$ using an EREW PRAM with $\mathcal{O}(n)$ processors.*

Proof: We use the algorithm for colouring a bounded degree graph presented by Goldberg and Plotkin in [30]. It runs in time $\mathcal{O}(\log^* n)$ using an EREW PRAM with $\mathcal{O}(n)$ processors. All steps inside the for-loop with exception of the colouring of G_i take constant time and the colouring of G_i can be done in time $\mathcal{O}(\log^* |U_i|)$. We know that $|U_i| < n$, so one iteration of the for-loop takes less than $\mathcal{O}(\log^* n)$ time. The theorem follows from the fact that Δ is a constant. □

2.4 Maximal f-dependent set problem

A generalization of the maximal k-dependent set problem is the f-dependent set problem. Given a positive integer function f defined on the set of vertices, a f-dependent set is a subset $\mathcal{F}$ of V such that each vertex $v \in \mathcal{F}$ is adjacent to at most $f(v)$ vertices in $\mathcal{F}$. Algorithm 2.2 can be easily generalized to construct a maximal f-dependent set in G provided an integer function f defined on the set of vertices of G is given. It is enough to redefine B as the set of all vertices in V or R respectively such that $f(v)$ minus the number of neighbours of v in Q is non-negative. By reasoning analogously as in the proof of Lemma 2.3.2 we conclude that the while block is iterated $\mathcal{O}(\max_{v \in V}(f(v))^2)$ times. Hence, we obtain the following generalization of Theorem 2.3.2 leaving the proof details to the reader.

Theorem 2.4.1 *Let G be a graph on n vertices and m edges, and let f be a positive integer function defined on the set of vertices of G. A maximal f-dependent set in G can be computed in time $\mathcal{O}((\log^3 n) \cdot (\max_{v \in V} f(v)^2))$ using an EREW PRAM with $\mathcal{O}(n^2)$ processors.*

Chapter 3

f-matchings

To match or not to match
What was the question?

Anon

Consider a graph $G = (V, E)$, and an integer function f defined on V. An f-matching in G is a subset of E such that for each vertex v at most $f(v)$ edges incident to v are in the subset. Note that a matching is simply a 1-matching. The *maximal, maximum* and *perfect f-matching* problems are very natural generalizations of the maximal, maximum and perfect matching problems respectively (see [5, 16]). An f-matching in G is maximal if it cannot be extended to any larger f-matching in G. An f-matching in G is maximum if it has the largest cardinality among all f-matchings in G. Finally, an f-matching in G is perfect if for each vertex v there are exactly $f(v)$ edges in the f-matching incident to v.

3.1 Maximum f-matching problem

The problem of constructing a maximum f-matching can be many-one reduced to the problem of constructing a maximum matching by generalizing Tutte's reduction of the problem of finding an f-factor (f-regular spanning subgraph) to the problem of finding an 1-factor (perfect matching) in a graph [53]. In effect, we obtain the following theorem.

Theorem 3.1.1 *For a graph $G = (V, E)$ on n vertices and a function $f : V \to \{1, ..., k\}$, we can construct a graph G' such that any maximum matching of G' yields a maximum f-matching of G. Both the construction of G' and the construction of the maximum f-matching of G on the basis of a maximum matching of G' can be accomplished in time $\mathcal{O}(\log n)$ on an EREW PRAM with $\mathcal{O}(\frac{(n+m)k}{\log n})$ processors, where $k < n$, n and m are respectively the number of vertices and edges of the input graph.*

Proof: Construct the graph $G' = (V', E')$, generalizing Tutte's construction [53], as follows. Set V' to $V_f \cup V_E$ where

1. $V_f = \{v_i \mid v \in V \ \& \ 1 \leq i \leq f(v)\}$ (there are $f(v)$ copies of v in G')

2. $V_E = \{v_e \mid \exists w \in V \ s.t. \ (v, w) = e \ \& \ e \in E\}$ (for each edge e incident to v the auxiliary vertex v_e is in G')

Next, set E' to $E_f \cup E_E$ where

1. $E_f = \{(v_i, v_e) \mid v_i \in V \ \& \ v_e \in V_E\}$ (each copy of v is adjacent to each auxiliary vertex induced by v and an adjacent edge e)

2. $E_E = \{(e_v, e_w) \mid (v, w) = e \ \& \ e \in E\}$ (two auxiliary vertices are adjacent if they are induced by the same edge)

Note that G' has $\mathcal{O}(nk)$ vertices and $\mathcal{O}(mk)$ edges. Consider a maximum matching M in G'. We may assume without loss of generality that for each edge d in E_E both its endpoints are incident to an edge in M (*). In other words, either $d = (v_e, w_e)$ is in M or for some unique i, j, the edges (v_i, v_e), (w_e, w_j) are in M. Otherwise, we can always insert d in M deleting the single edge in M incident to an endpoint of d so the resulting matching remains maximum. Set M_f to $\{e \mid e = (v, w) \ \& \ (e_v, e_w) \notin M\}$. Consider a vertex v in G. For each edge e in M_f incident to v there exists a unique i such that (v_i, v_e) is in M by the assumed property (*) of M and the definition of M_f. As there are $f(v)$ copies v_i of v in G' and M is a matching of G', the set M_f is an f-matching of G. Also, by the definition of M_f and (*), we have $|M| = |M_f| + |E|$.

Contrary, given an f-matching B in G, we can easily build a matching B_1 of G' in two stages. In the first stage for each vertex v in G we

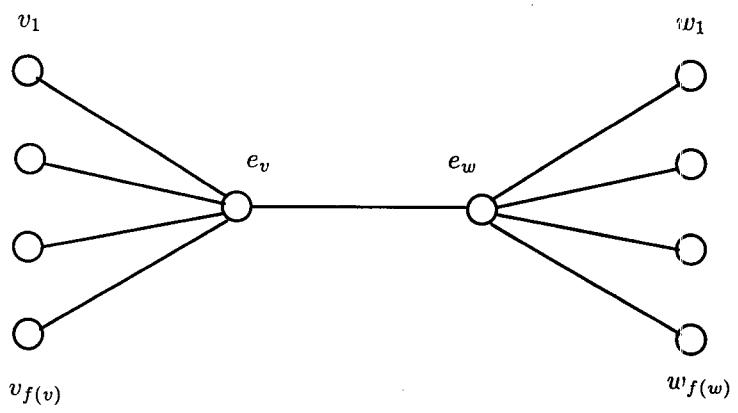


Figure 3.1: The subgraph of G' corresponding to an edge $e = (v, w)$ of G.

number edges in B incident to v, and for such an i-th edge e, we insert (v_i, v_e) into B_1. In the second stage for each edge $e = (v, w) \in E - B$ we insert (e_v, e_w) in B_1. It is easy to see that so constructed B_1 is a matching of G' with $2|B| + (|E| - |B|)$ edges, i.e., $|B_1| = |B| + |E|$. It follows from the maximum cardinality property of M that $|M| \geq |B| + |E|$. Thus M_f is a maximum f-matching by $|M| = |M_f| + |E|$

The construction of the graph G' on the basis of G and f, and the construction of the f-matching M_f on the basis of a maximum matching M can be done within the time and processor bounds specified in the theorem, among others by applying an optimal logarithmic-time EREW PRAM algorithm for list ranking to adjacency lists [36]. □

A maximum matching in a graph on n vertices, and m edges can be constructed sequentially in time $\mathcal{O}(\sqrt{n}m)$ [48]. In parallel, it can be constructed in time $\mathcal{O}(\log^2 n)$ using a randomized PRAM with $\mathcal{O}(nM(n)m)$ processors [49], or in time $\mathcal{O}(\log^3 n)$ using a randomized PRAM with $\mathcal{O}(nM(n))$ processors [19] ($M(n)$ is the number of arithmetic operations used by the best known sequential algorithm for multiplying two $n \times n$ matrices; currently $M(n) = \mathcal{O}(n^{2.376})$). The above facts combined with Theorem 3.1.1 and the estimation of the size of G' yield the following corollary.

Corollary 3.1.1 *A maximum f-matching in a graph on n vertices and m edges can be constructed sequentially in time $\mathcal{O}(n^2m)$, and in parallel in time $\mathcal{O}(\log^2 n)$ using a randomized PRAM with $\mathcal{O}(n^3M(n^2)m)$ processors, or in time $\mathcal{O}(\log^3 n)$ using $\mathcal{O}(n^2M(n^2))$ processors on a randomized PRAM.*

3.1.1 The decision version of DSP is in NC

The so called *degree sequence problem* (DSP for short) can be seen as the restriction of the perfect f-matching problem to the complete graph case. Independently, it can be defined as follows: for a sequence $d_1,...,d_n$ of natural numbers, construct if possible a simple graph on n nodes such that the degree of the i:th node is d_i. We shall call $d_1,...,d_n$ a *degree sequence* if such a graph exists. The degree sequence problem is considered as a fundamental problem in graph theory [2, 5]. It admits a trivial greedy polynomial-time algorithm which can be refined to a linear one (e.g., see Section 2 in [2]). Also, if maximum connectivity requirements on the graph to construct are added it still can be solved in nearly linear time [2].

The degree sequence problem has a solution if the integers d_i satisfy the following elementary inequalities for $k = 1,..n$ due to Erdös and Gallai [5, 18]:

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{j=k+1}^{n} min\{k, d_j\}$$

In [2], Takao Asano has recently observed that the computationally simpler inequalities corresponding to the case where the graph to construct is required k-connected, $k \geq 1$, are NC testable. The presence of *min* makes the problem of testing in our non-necessarily connected case a bit more difficult. Nevertheless we can report the following optimal result.

Theorem 3.1.2 *One can decide whether a sequence of integers $d_1 \geq d_2 \geq \cdots \geq d_n$ is a degree sequence in time $O(\log n)$ using $O(\frac{n}{\log n})$ EREW PRAM processors.*

Proof: We may assume w.l.o.g. that the integers $d_1, ..., d_n$ are in the range $[1, n-1]$. Compute the prefix sums PR_j, and the postfix sums PO_j,

$j = 1, ..., n$, for the sequence $d_1, ..., d_n$. Form a sequence $a_1, ..., a_{n-1}$ of integers such $a_i = i$ for $i = 1, ..., n-1$, and merge it with the sequence $d_1, ..., d_n$ in such a way that if $d_j = a_k$ then d_j precedes a_k. Assign to each d_j element weight 1 and to each a_k element weight 0 and compute the weighted ranks R_k for each a_k element. Now the inequalities due to Erdös and Gallai can be rephrased as follows:

$$PR_k \leq k(k-1) + \max\{(R_k - k)k, 0\} + PO_{\max\{R_k+1, k+1\}}$$

Thus, assuming the prefix sums, postfix sums and weighted ranks are computed, the inequalities can be easily checked in logarithmic time using $O(\frac{n}{\log n})$ EREW PRAM processors. The prefix sums, postfix sums and weighted ranks can be computed within the above resource bounds by using the known work-optimal EREW PRAM algorithms for prefix sums, merging, and weighted list ranking respectively [9, 36]. □

3.2 Maximal f-matching for restricted graphs

The computation of a maximal f-matching is generally more difficult than that of a maximal matching. Obviously, a maximal f-matching can be computed in linear time by a greedy sequential algorithm. Unfortunately such an algorithm is not well parallelizable (see [47]).

In this section we present two parallel algorithms for maximal f-matching. The first one applies to graphs of bounded valence and it is a simple reduction to the problem of constructing a maximal matching in graphs of bounded valence. It can be implemented in time $\mathcal{O}(\log^* n)$ on an EREW PRAM with a linear number of processors [16]. The second algorithm constructs a maximal f-matching in graphs with arboricity bounded by a constant. It can be implemented in time $\mathcal{O}(\log^2 n)$ on an EREW PRAM with $\mathcal{O}(\frac{n}{\log n})$ processors [21].

3.2.1 Maximal f-matching in constant-degree graphs

A maximal matching in a graph of bounded valence can be computed by reduction to the problem of finding a maximal independent set in the corresponding edge graph (which is also of bounded valence) in time

```
Algorithm  Maximal-f-Matching(G, f)
input :    A bounded valence graph G = (V, E),
           its matching function f.
output :   A maximal f-matching M of G.
method :
           M ← ∅;   G_0 = (V_0, E_0) ← G = (V, E);
           f_0 ← f;   i ← 0;
           while E_i ≠ ∅ do
                 U ← {v ∈ V_i | f_i(v) > 0};
                 F ← {(u, v) | u, v ∈ U} ∩ E_i;
                 Max ← Maximal-1-Matching((U, F));
                 M ← M ∪ Max;
                 i ← i + 1;
                 (V_i, E_i) ← (U, F \ Max);
                 for each u ∈ U in parallel do
                     if u is incident to an edge in Max then
                        f_i(u) ← f_{i-1}(u) − 1
                     else
                        f_i(u) ← f_{i-1}(u);
                 endfor
           endwhile
           output M;
end  Maximal-f-Matching
```

ALGORITHM 3.1

$\mathcal{O}(\log^* n)$ on an EREW PRAM with $\mathcal{O}(n)$ processors [30]. In the Algorithm 3.1 we use this fact to achieve the same asymptotic resource-bounds for maximal f-matching in the bounded valence case.

Theorem 3.2.1 *Let G be a graph of bounded valence on n vertices. A maximal f-matching in G can be found in time $\mathcal{O}(\log^* n)$ on an EREW PRAM with $\mathcal{O}(n)$ processors.*

Proof: All steps of the while loop except the computation of the maximal matching *Max* take constant time on an EREW PRAM with n processors. The computation of *Max* can be done in time $\mathcal{O}(\log^* n)$ using an EREW PRAM with $\mathcal{O}(n)$ processors [30]. Observe that if (u, v) is an edge in the graph $G_i = (V_i, E_i)$ before the i:th iteration of the while loop then either at least one of vertices u, v does not appear in

the graph $G_{i+1} = (V_{i+1}, E_{i+1})$ or the sum of the degrees of u and v is at least 1 smaller than the sum of their degrees in G_i. We conclude that the number of the iterations of the while statement is at most 2 times the valence of the input graph G. □

3.2.2 f-matching in sparse graphs

The following fact due to Osiakwan and Akl [50] yields a $\mathcal{O}\log(n)$-time optimal EREW PRAM algorithm for maximal f-matching in forests.

Fact 3.2.1 *Let $T = (V, E)$ be a tree with non-negative integer vertex capacities given by the function f defined on V. A maximum cardinality f-matching of T can be found in time $\mathcal{O}(\log n)$ using $\mathcal{O}(\frac{n}{\log n})$ EREW PRAM processors.*

We can use the method for finding a maximal f-matching in a forest implied by Fact 3.2.1 to find maximal f-matchings in graphs of constant arboricity. Recall that a graph G is said to be of arboricity c if and only if it can be decomposed into c disjoint spanning forests.

Theorem 3.2.2 *Let $G = (V, E)$ be a constant arboricity graph with non-negative integer vertex capacities given by the function f defined on V. A maximal f-matching of G can be found in time $\mathcal{O}(\log^2 n)$ using an EREW PRAM with $\mathcal{O}(\frac{n}{\log n})$ processors.*

Proof: The reduction to the corresponding problem for a forest is as follows. First, in parallel, for each vertex v in V, we choose the edge $e(v)$ which connects v with its neighbour of highest number. Next, we extract from G the subgraph F spanned by the edges $e(v)$. By using the standard logarithmic-time methods for finding the maximum and grouping elements into logarithmic-size blocks assigned to single processors (see [27]) these two steps can be done in time $\mathcal{O}(\log n)$ using an EREW PRAM with $\mathcal{O}(\frac{n}{\log n})$ processors. It is not difficult to see that F is a spanning forest for G. Now, we use the method implied by Fact 3.2.1 to find a maximal f-matching M in F. M is our initial f-matching in G which we shall augment in subsequent iterations to obtain a maximal f-matching of G. Note that none of the edges in F outside M could be used to augment M. Therefore, we can remove all

the edges of F from G, appropriately decrease the capacities of vertices in G by the number of incident edges in M, and iterate our method on the resulting subgraph of G, *etc.* During the consecutive iterations we augment M with the produced maximal f-matchings of the consecutive forests. By induction on the number of iterations we deduce that the current f-matching M cannot be augmented by any edge deleted from G in this or the previous iterations. As the number of edges in a graph of arboricity bounded by c is bounded from above by c times the number of its non-isolated vertices and, on the other hand, the number of edges in a spanning forest of a graph is at least half the number of the non-isolated vertices of the graph, we conclude the following: after a logarithmic number of iterations G is reduced to a set of isolated vertices. Thus, the final f-matching M is maximal. By Fact 3.2.1, each of the iterations takes time $\mathcal{O}(\log n)$ on a EREW PRAM with $\mathcal{O}(\frac{n}{\log n})$ processors which implies the thesis. □

Corollary 3.2.1 *A maximal f-matching in a graph of bounded genus, in particular planar graphs, can be found in time $\mathcal{O}(\log^2 n)$ using an EREW PRAM with $\mathcal{O}(\frac{n}{\log n})$ processors.*

3.3 f-matching in general graphs

In this section we present the first deterministic NC parallel algorithm for constructing a maximal f-matching in the general case.

The algorithm due to Israeli and Shiloach constructs a maximal matching in a graph on n vertices and m edges in time $\mathcal{O}(\log^3 n)$ on an CRCW PRAM with $\mathcal{O}(n+m)$ processors [34]. Our algorithm is an advanced generalization of the Israeli and Shiloach's algorithm to include maximal f-matching achieving the same asymptotic resource-bounds. The generalized algorithm consists of several procedures. In order to present them we need the following notation.
For each vertex $v \in V$ we define its weight $w(G, v, f)$ with respect to f as follows:

$$w(G, v, f) = \min_{s \in \mathcal{N}} \{s \mid 2^s f(v) \geq \deg_G(v)\}.$$

Let $W(G, f) = \max_{v \in V} w(G, v, f)$. The number $W(G, f)$ will be called the weight of the graph G with respect to f (see Fig 3.2).

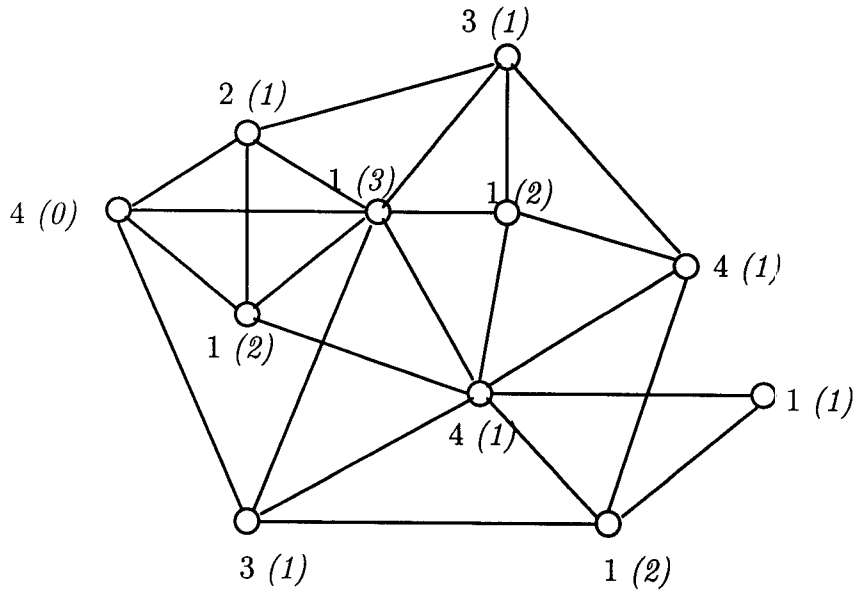


Figure 3.2: An example of a graph G with capacities $f(v)$ and vertex weights $w(G, v, f)$. The weights are in italic, The weight of the graph is 3.

Observe that if $W(G, f) = 0$ then all edges of the graph belong to a maximal f-matching.
A vertex v is said to be an *active* one in G with respect to a matching function f if and only if $2^{W(G,f)-1}f(v) \leq \deg_G(v) \leq 2^{W(G,f)}f(v)$. An active vertex v is called *safe* if $\deg_G(v) = 2^{W(G,f)-1}f(v)$. The procedure REDUCE, reduces G (removing some of its edges) to a graph G' such that $W(G', f) \leq W(G, f) - 1$.

Lemma 3.3.1 *If* $W(G, f) \geq 1$ *then* $W(G', f) \leq W(G, f) - 1$.

Proof: First of all let us observe that if $w(G, v, f) \leq W(G, f) - 1$, for some $v \in V$, then also $w(G', v, f) \leq W(G, f) - 1$.
Consider now a vertex v for which $w(G, v, f) = W(G, f)$. Then $2^{W(G,f)-1}f(v) < \deg_G(v) \leq 2^{W(G,f)}f(v)$ and v is a non-safe vertex. Hence $\deg_{G'}(v) \leq \lceil \frac{1}{2} \deg_G(v) \rceil \leq 2^{W(G,f)-1}f(v)$. It implies $W(G', f) \leq W(G, f) - 1$. □

Procedure *REDUCE*(G, f, G')
input : a simple graph $G = (V, E)$.
a matching function $f : V \to \mathcal{N}$ such that
$W(G, f) \geq 1$.
(* We assume that $\deg_G(v), w(G, v, f), \quad \forall v \in V$, *)
(* and $W(G, f)$ are computed. *)
output : a subgraph $G' = (V, E')$ of the graph G such that
$W(G', f) \leq W(G, f) - 1$.
method :

Construct an auxiliary graph H induced by all those edges of G for which at least one end-point is an active vertex (called later real edges);
Compute connected components of H;
Label edges of each connected component 1 if all its active vertices are safe. Call such components safe;
Let H' be a subgraph of H containing only non-safe components;
for every vertex of odd degree in H'
add an edge to an introduced vertex u;
Find an Eulerian circuit in each connected component C of H';
Label the edges of each component C of H' with 0 and 1 in the following way:
(1) **if** C contains the introduced vertex u **then**
starting at u trace the Eulerian cycle and label the edges 0 and 1 alternately;
if the number of real edges labeled 0 is larger than the number of real edges labeled 1 **then**
exchange 0's for 1's and 1's for 0's;
(2) **if** C does not contain u **then**
find in C a vertex x which is a neighbour of a non-safe vertex y;
Starting from the edge (x, y) trace the cycle and label the edges 1 and 0 alternately ((x, y) is labeled 1);
if the length of the cycle is odd **then**
exchange the label of (x, y) for 0;
$G' \leftarrow$ the graph obtained by the removal of all real edges labeled with 0 from G ;
output G';
end *REDUCE*

PROCEDURE REDUCE

Lemma 3.3.2 *Let $W(G,f) \geq 2$. If v is an active vertex in G then v remains active in G'.*

Proof: It suffices to show that if v is an active vertex in G then $2^{W(G,f)-2}f(v) \leq \deg_{G'}(v) \leq 2^{W(G,f)-1}f(v)$. Let us consider three cases:

1. v belongs to a connected component of H in which all active vertices are safe. Then
$$\deg_{G'}(v) = \deg_G(v) = 2^{W(G,f)-1}f(v).$$
2. v is in the same connected component of H' as the introduced vertex u. The procedure REDUCE reduces the input graph in such a way that
$$\lfloor \frac{1}{2}\deg_G(v) \rfloor \leq \deg_{G'}(v) \leq \lceil \frac{1}{2}\deg_G(v) \rceil.$$
Hence
$$2^{W(G,f)-2}f(v) \leq \deg_{G'}(v) \leq 2^{W(G,f)-1}f(v).$$
3. v is in a connected component of H' not containing u. If $v \neq y$ (see description of the procedure) then
$$\deg_{G'}(v) = \frac{\deg_G(v)}{2}.$$
If $v = y$ then v is a non-safe vertex. Hence
$$\deg_G(v) \geq 2^{W(G,f)-1}f(v) + 2.$$

If the Eulerian circuit in the connected component contained v has odd length then
$$\deg_{G'}(v) = \frac{\deg_G(v)}{2} - 1 \geq 2^{W(G,f)-2}f(v)$$
otherwise
$$\deg_{G'}(v) = \frac{\deg_G(v)}{2}$$
Hence always
$$2^{W(G,f)-2}f(v) \leq \deg_{G'}(v) \leq 2^{W(G,f)-1}f(v). \qquad \square$$

We define a procedure f-MATCHING which computes some "large" f-matching M for a given input graph G.

```
Procedure  f-MATCHING(G, f, M)
input :    a graph G = (V, E).
           a matching function f : V → N
output :   a large f-matching for G.
method :
           i ← 0;
           G_0 ← G;
           repeat
                  for all vertices v ∈ V in parallel do
                      compute deg_{G_i}(v);
                      compute w(G_i, v, f);
                  endfor
                  compute W(G_i, f);
                  j ← i;
                  if W(G_i, f) ≥ 1 then
                     REDUCE(G_i, f, G_{i+1});
                     i ← i + 1;
                  endif
           until j = i;
           M ← the set of edges of the graph G_i;
end  f-MATCHING
```

PROCEDURE f-MATCHING

Lemma 3.3.3 *The set of edges M computed by the procedure* f-MATCHING *is an f-matching in the input graph G.*

Proof: Let $i_{\max}$ be the maximal value of the variable i. The weight of the graph $G_{i_{\max}}$ with respect to the function f is 0. Then for each vertex $v \in V, \deg_{G_{i_{\max}}}(v) \leq f(v)$. This implies that M is an f-matching in G. □

Let M be an f-matching in a graph G. The procedure MODIFY, deletes the edges of M from the graph producing a graph K and next computes a matching function h such that any maximal h-matching of K extended with the edges of the set M is a maximal f-matching in G:

Lemma 3.3.4 *Let H be a maximal h-matching in K then $H \cup M$ is a maximal f-matching in G.*

Proof: It is sufficient to observe that:

```
Procedure  MODIFY(G, f, M, K, h)
input :    a graph G = (V, E),
           its matching function f,
           an f-matching M in G.
output :   a subgraph K = (V, E') of G,
           a matching function h for K such that any
           maximal h-matching H extended with the edges
           of M is a maximal f-matching of the graph G.
method :
           M' ← {(u, v) ∈ E | (u, v) ∈ M or u is incident to f(u)
                 edges in M or v is incident to f(v) edges in M};
           E' ← E \ M';
           K ← (V, E');
           for v ∈ V in parallel do
               if deg_K(v) = 0 then
                  h(v) ← 1;
               else
                  h(v) ← f(v) − |{(v, u) | (v, u) ∈ M}|;
               endif
           endfor
           output K, h;
end MODIFY
```

PROCEDURE MODIFY

- E' consists of all possible edges which can extend the f-matching M, and

- for each non-isolated vertex v in H its new matching value $h(v)$ is equal to the old matching value $f(v)$ decreased by the number of edges belonging to the f-matching M. □

Now we are ready to write the entire algorithm for finding maximal f-matchings in graphs:

Consider a graph G and its matching function f. Let A be a subset of the set of vertices of the graph. By $\mathrm{cost}(G, A, f)$ we will denote a cost of the set A with respect to the function f defined as follows:

$$\mathrm{cost}(G, A, f) = \sum_{\substack{u \text{ is non-isolated} \\ \text{in } A}} f(u).$$

Algorithm *Maximal-f-Matching*(G, f)
input : a graph $G = (V, E)$,
a matching function $f : V \to \mathcal{N}$.
output : a maximal f-matching *MaxM* in the graph G.
method :
$i \leftarrow 0$;
$G_0 = (V_0, E_0) \leftarrow G = (V, E)$;
$f_0 \leftarrow f$;
$MaxM \leftarrow \emptyset$;
while $|E_i| > 0$ **do**
f-MATCHING(G_i, f_i, M_i);
$MaxM \leftarrow MaxM \cup M_i$;
MODIFY$(G_i, f_i, M_i, G_{i+1}, f_{i+1})$;
$i \leftarrow i + 1$;
endwhile
end *Maximal-f-Matching*

ALGORITHM 3.2

If A is empty or contains only isolated vertices, we assume that: $\text{cost}(G, A, f) = 0$.

Lemma 3.3.5 *Let G be a graph, f its matching function and C a vertex cover of the graph G. Next, let M be the f-matching which is the result of the call f-*MATCHING*(f, G, M). Finally, let K, h be the graph and its matching function, respectively, obtained as the result of the call* MODIFY*(G, f, M, K, h). Then there exists a vertex cover A in the graph K such that $\text{cost}(K, A, h) \leq \frac{5}{6}\text{cost}(G, C, f)$.*

Proof: Let us consider the application of the procedure f-MATCHING to the graph G. Let $i_{\max}$ be the maximum value of the variable i. If $i_{\max} = 0$ then all edges of the graph belong to M. In this case it suffices to take as the set A simply the empty set. Let us assume now that $i_{\max} > 0$. Let B denote a set of all active vertices in the graph $G_{i_{\max}-1}$. Each edge $e \in E$ has either both endpoints in $V \setminus B$ or at least one of its endpoints belongs to B. If both endpoints belong to $V \setminus B$ then naturally e is an edge in the graph $G_{i_{\max}}$ and hence it is in M. This and the fact that each active vertex in G_j, for each $j < i_{\max-1}$, remains active in $G_{i_{\max-1}}$ (Lemma 3.3.1) imply that B is a vertex cover of the

graph K. Let us observe that

$$\text{cost}(G,B,f) = \sum_{u\in B} f(u) \leq \sum_{u\in B} \deg_{G_{i_{\max-1}}}(u).$$

Let ℓ denote the number of edges in $G_{i_{\max-1}}$ with exactly one end-point in B and k the number of edges with both end-points in B. Then $\text{cost}(G,B,f) \leq \ell+2k$. The procedure REDUCE reduces the graph $G_{i_{\max-1}}$ in such a way that the f-matching M contains at least $\frac{1}{3}(\ell+k)$ edges incident with vertices in B. Each connected component of H in which all active vertices are safe gives all its edges to M. The connected component in H' containing the introduced vertex u gives at least half of its real edges to M. If a connected component C of H' does not contain u and has e edges then it gives $\lfloor \frac{e}{2} \rfloor \geq \frac{e}{3}$ edges to M ($e > 1$, because C contains a safe active vertex). Since $\frac{1}{3}(\ell+k) \geq \frac{1}{6}(\ell+2k)$, we have the following:

$$\begin{aligned}
\text{cost}(K,B,h) &\leq \text{cost}(G,B,f) - \tfrac{1}{3}(\ell+k) \\
&\leq \text{cost}(G,B,f) - \tfrac{1}{6}(\ell+2k) \\
&\leq \text{cost}(G,B,f) - \tfrac{1}{6}\text{cost}(G,B,f) \\
&\leq \tfrac{5}{6}\ \text{cost}(G,B,f).
\end{aligned}$$

Let us consider now two cases:

CASE 1 : $\text{cost}(G,B,f) \leq \text{cost}(G,C,f)$. If we take as the set A the set B then
$\text{cost}(K,A,h) \leq \frac{5}{6}\text{cost}(G,C,f)$.

CASE 2 : $\text{cost}(G,B,f) > \text{cost}(G,C,f)$. Let us observe that M contains at least $\frac{1}{6}\text{cost}(G,B,f)$ edges.
If we take C as the set A the following holds:

$$\begin{aligned}
\text{cost}\ (K,A,h) &\leq \text{cost}\ (G,C,f) - \tfrac{1}{6}\text{cost}(G,B,f) \\
&\leq \text{cost}\ (G,C,f) - \tfrac{1}{6}\ \text{cost}\ (G,C,f) \\
&\leq \tfrac{5}{6}\ \text{cost}\ (G,C,f).
\end{aligned}$$

□

Theorem 3.3.1 *Let $G = (V,E)$ be an n-vertex graph with m edges and let f be its matching function. A maximal f-matching in G can be computed in time $\mathcal{O}(\log^3 n)$ using a CRCW PRAM with $\mathcal{O}(n+m)$ processors or in time $\mathcal{O}(\log^4 n)$ using an EREW PRAM with $\mathcal{O}(n+m)$ processors.*

Proof: We can compute a maximal f-matching using the algorithm MAXIMAL-f-MATCHING. It follows directly from Lemmata 3.3.3 and 3.3.4 that if the algorithm stops then *MaxM* is a maximal f-matching in G. We show how to implement efficiently the algorithm MAXIMAL-f-MATCHING. We assume that the input graph G is represented by adjacency lists. The complexity of the procedure REDUCE depends on the complexity of computing connected components and Eulerian cycles. This can be done in time $\mathcal{O}(\log n)$ on an $\mathcal{O}(n+m)$-processor CRCW PRAM or in time $\mathcal{O}(\log^2 n)$ on an EREW PRAM (cf. [3, 52]). Each iteration of the repeat loop in the procedure f-MATCHING consists of some computations on the adjacency lists and of the call of the procedure REDUCE. Hence it takes time $\mathcal{O}(\log n)$ or $\mathcal{O}(\log^2 n)$ depending on the model of computations. It follows from Lemma 3.3.1 that the number of iterations of the repeat loop can not be larger than $\lceil \log(n-1) \rceil$. Hence the procedure f-MATCHING runs in time $\mathcal{O}(\log^2 n)$ or $\mathcal{O}(\log^3 n)$ using only $\mathcal{O}(n+m)$ processors. The procedure MODIFY consists only of simple computations on the adjacency lists. It takes $\mathcal{O}(\log n)$ time. Let us observe that the cost of each vertex cover in the input graph G is bounded by n^2 from above. Taking into account Lemma 3.3.5 we infer that the number of iterations of the while loop of the algorithm is $\mathcal{O}(\log n)$. Hence the algorithm stops and runs in time $\mathcal{O}(\log^3 n)$ or $\mathcal{O}(\log^4 n)$, depending on the model of computation, and uses only the processors associated with the vertices and the edges of the graph. □

The above theorem yields immediately the following corollary.

Corollary 3.3.6 *The problem of computing a maximal f-matching is in NC.*

3.4 A randomized parallel algorithm for maximal f-matchings

In this section we present a RNC-algorithm for maximal f-matchings in the general case.

For many problems the main advantage of randomization is gaining simplicity. This happens in the computation of maximal matchings

in graphs. We show that a known and quite simple randomized algorithm for maximal matchings has a natural extension to the so called f-matchings.

The whole structure of our randomized algorithm is very simple: it finds a partial f-matching F, deletes the edges of F, updates capacities and then removes edges incident to vertices with actual capacity equal to zero. This continues until the graph is empty.

The main operation is to find a large partial f-matching F. This is done by a kind of *handshaking strategy* where a number of edges are marked by their incident vertices and some of the marked edges are selected by the neighbours of the vertices which marked them. The set of selected vertices called F is almost always an f-matching.

Note that the set F of edges added in one iteration of the repeat-block of Algorithm 3.3 is a partial f-matching. If $f(v) \geq 2$, then, just before the cleanup stage, the number of edges of F incident to v is at most $\lceil \frac{f(v)}{2} \rceil + \lfloor \frac{f(v)}{2} \rfloor = f(v)$. Next, if $f(v) = 1$ then at most 2 edges of F are incident to v after 'bound-indegrees'. In the latter case, if exactly 2 edges are incident to v at least one of them will be removed in the repair stage.

The analysis of the algorithm is based on a simple combinatorial property of undirected graphs. Let G be an undirected graph. Associate with each vertex v of G a number $\varphi(v)$. The vertex v is *good* if and only if at least $\frac{1}{3}$ of its neighbours w satisfy $\varphi(v) \leq \varphi(w)$. The edge is called *good* if and only if at least one of its endpoints is good. Vertices which are not good are called *bad* vertices and edges which are not good are called *bad* edges.

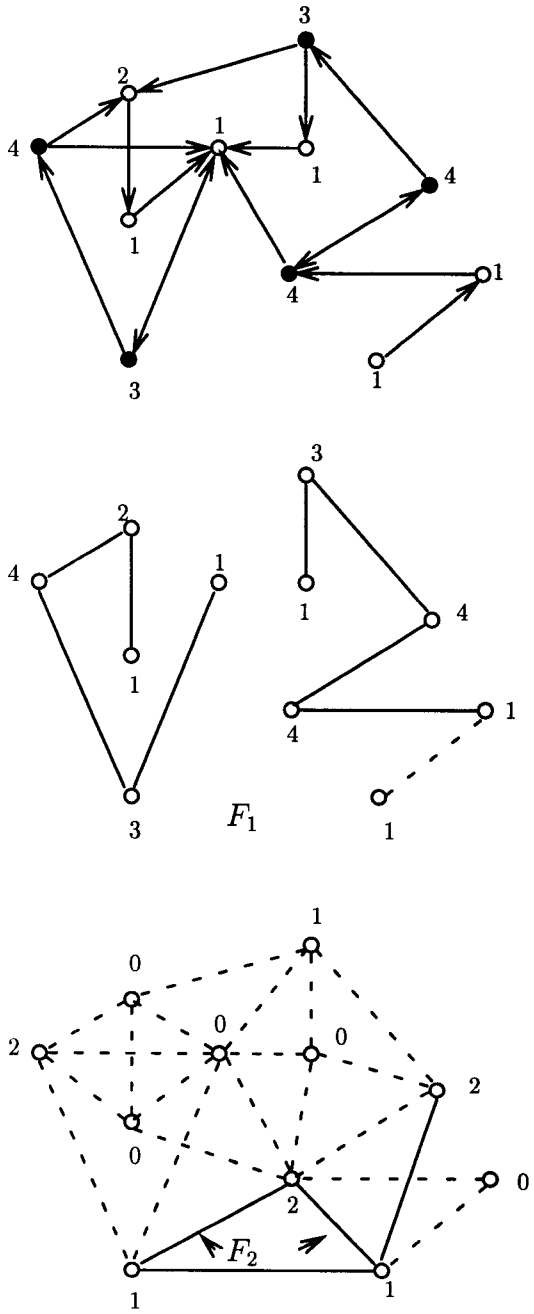
Lemma 3.4.1 *At least half of the edges of G are good.*

Proof: Let us direct all edges in the graph. If (v, w) is an undirected edge such that $\varphi(v) > \varphi(w)$ we put the direction from v to w, otherwise we fix any (unique) direction for (v, w). By the definition, all bad edges end in bad vertices and each bad vertex has indegree not greater than $\frac{1}{2}$ times its outdegree, so we have:

$$\begin{aligned} m &= \sum_{v \in V} \text{outdeg}(v) \\ &\geq \sum_{\substack{v \text{ is bad} \\ v \in V}} \text{outdeg}(v) \end{aligned}$$

```
Algorithm MaxMatch(G, f)
input :  A graph G = (V, E) and its matching function f.
output : A maximal f-matching M of G.
method :
        M ← ∅;
        repeat
            choose-edges:
                for each vertex v in parallel do
                    Assign processors p(v)_i  i = 1, 2, ..., ⌈½f(v)⌉
                    to v;
                    for i ← 1 ... ⌈½f(v)⌉ in parallel do
                        p(v)_i marks a random edge incident to v;
                    endfor
                endfor
            bound-indegrees:
                for each vertex v in parallel do
                    S(v) ← the set of edges incident on v and
                           marked by its neighbours;
                    if f(v) = 1 then
                       v selects min{|S(v)|, 1} edges in S(v);
                    else if f(v) ≥ 2 then
                       v selects min{|S(v)|, ⌊½f(v)⌋} edges in S(v);
                    endif
                    F ← the set of selected edges;
                endfor
            repair:
                for each vertex v in parallel do
                    if f(v) = 1 and (v has two incident edges
                    in F) then
                       Randomly delete one of them from F;
                endfor
                { F is now an f-matching }
                M ← M ∪ F;
                E ← E \ F;
            cleanup:
                for each vertex v in parallel do
                    if deg_{γ(F)}(v) = f(v) then
                       remove all edges incident to v from E;
                    f(v) ← min{deg_{γ(E)}(v), f(v) − deg_{γ(F)}(v)}
                endfor
        until G has no edges;
        output M;
end MaxMatch
```

ALGORITHM 3.3



a) The edges marked in step "choose-edges", the edge is directed out of the vertex which has marked it. Good vertices are drawn here in white.

b) The edges selected in the step "bound-indegrees", the dashed edge is removed in the step "cleanup". F_1 is the set of nondashed edges.

c) The graph G and its capacities after the first iteration of the algorithm MaxMatch The set F_1 of edges is removed, the capacities are updated and edges incident to zero capacity vertices are removed. The removed edges are dashed. The next value of F is the set F_2 consisting of edges in bold. The final f-matching is the union of F_1 and F_2.

Figure 3.3: An example of the first iteration of the algorithm MaxMatch.

$$\geq 2 \cdot \sum_{\substack{v \text{ is bad} \\ v \in V}} \text{indeg}(v)$$
$$\geq 2 \cdot \text{number of bad edges.}$$

□

The crucial point in our algorithm is to reduce capacities for a big proportion of the so called good edges by a constant fraction in each iteration. This guarantees the logarithmic number of iterations since at least half of the edges of the graph are always good.

Here we take $\varphi(v) = \frac{f(v)}{\deg(v)}$. The lemma below says that a good vertex is likely to have its capacity reduced by a constant factor.

Lemma 3.4.2 *Consider a single execution of the repeat-block. The probability that a good vertex v of positive degree in G and positive capacity has its capacity reduced by at least $\frac{f(v)}{53}$ is greater than some positive constant.*

Proof: Let X, Y and Z denote the number of edges of F incident to v after the choose-edges stage, the bound-indegrees stage and the repair stage, respectively. We need to show $Pr[Z \geq \frac{f(v)}{53}] > c > 0$ for some constant c. The first step will be to show that $Pr[X \geq \frac{f(v)}{13}] > c' > 0$ for some constant c'.

Let v be a good vertex and let $d = d(v)$. Fix $\lceil \frac{d}{3} \rceil$ neighbours $v_1, \ldots, v_k$ of v such that $\frac{f(v)}{d} \leq \frac{f(v_i)}{\deg(v_i)}$ for $i = 1, \ldots, k$. Let M_i denote the event that v_i marks the edge (v_i, v) in the choose-edges stage.

We have $Pr[M_i] = 1 - (1 - \frac{1}{deg(v_i)})^{\lceil \frac{f(v_i)}{2} \rceil}$. For $0 < x, y < 1$ we have $(1-x)^y < e^{-xy} < 1 - \frac{xy}{2}$. Thus we get $Pr[M_i] > \frac{\lceil \frac{f(v_i)}{2} \rceil}{2deg(v_i)} \geq \frac{f(v)}{4d}$.

We conclude that $Pr[X \geq \frac{f(v)}{13}] \geq Pr[Q \geq \frac{f(v)}{13}]$ where Q has binomial distribution $B(\lceil \frac{d}{3} \rceil, \frac{f(v)}{4d})$.

In [31] the following variant of Chernoffs bound has been derived: Let $X_1, X_2, \ldots, X_k$ be independent 0-1 variables, and let M be the expected value of $S = X_1 + X_2 + \ldots + X_k$. Then

$$\Pr(S \leq (1-\epsilon)M) \leq \left(\frac{\exp(\epsilon)}{(1+\epsilon)^{1+\epsilon}} \right)^M \quad 0 \leq \epsilon \leq 1.$$

In our case we have $M \geq \frac{f(v)}{12}$. By substituting $\epsilon = \frac{1}{13}$ and observing that

$$\left(\frac{exp(\epsilon)}{(1+\epsilon)^{1+\epsilon}}\right) < 1 \tag{3.1}$$

we see that with $Pr[X \geq \frac{f(v)}{13}] > c' > 0$ for a constant c'. Now note that $Pr[Y \geq \frac{f(v)}{13}] = Pr[X \geq \frac{f(v)}{13}]$ and hence $Pr[Y \geq \frac{f(v)}{13}] > c' > 0$ for a constant c'.

Next we observe that $Pr[Z \geq \frac{f(v)}{53}] \geq Pr[Z \geq \frac{f(v)}{53} \wedge Y \geq \frac{f(v)}{13}] \geq c' Pr[Z \geq \frac{f(v)}{53} | Y \geq \frac{f(v)}{13}]$. Thus it suffices to show that

$$Pr[Z \geq \frac{f(v)}{53} | Y \geq \frac{f(v)}{13}] > c'' > 0 \tag{3.2}$$

for some constant c''. To see this, note that each edge (v_i, v) survives the repair stage independently with probability at least $\frac{1}{4}$. Thus $Pr[Z \geq \frac{f(v)}{53} | Y \geq \frac{f(v)}{13}] \geq Pr[Q \geq \frac{f(v)}{53}]$ where Q has binomial distribution $B(\lceil \frac{f(v)}{13} \rceil, \frac{1}{4})$. Now the relation (3.2) is established in the same manner as above using the Chernoff bound. This completes the claim of the lemma. □

Lemma 3.4.3 *The expected number of iterations of the block under the repeat instruction in Algorithm 3.3 is* $\mathcal{O}(\log^2 n)$.

Proof: Let e denote the number of edges of the current graph G in Algorithm 3.2. To prove the lemma it is enough to show that there exists a constant d such that after $d\lceil \log n \rceil$ iterations of the block in Algorithm 3.2, the expected number of edges in G is no greater than $\frac{1}{2}e$. The proof of this fact, where d is specified later, is as follows.

Suppose that after $d\lceil \log n \rceil$ iterations the expected number of edges in G is at least $\frac{1}{2}e$. Consider any of the above iterations. An edge in the current G in the iteration is said to be *very good* if it is good and the capacity of at least one of its good endpoints is decreased at least by $\frac{1}{53}$:rd. By Lemma 3.4.1 at least $\frac{e}{4}$ edges are good in this iteration of the repeat-block. By Lemma 3.4.2 a positive fraction of these, say be edges, are very good. Therefore during the $d\lceil \log n \rceil$ iterations, the expected number of occurrences of very good edges is at least $bed\lceil \log n \rceil$.

On the other hand, there exists a constant c such that a vertex v can occur as a good vertex, which has a positive capacity and looses at least

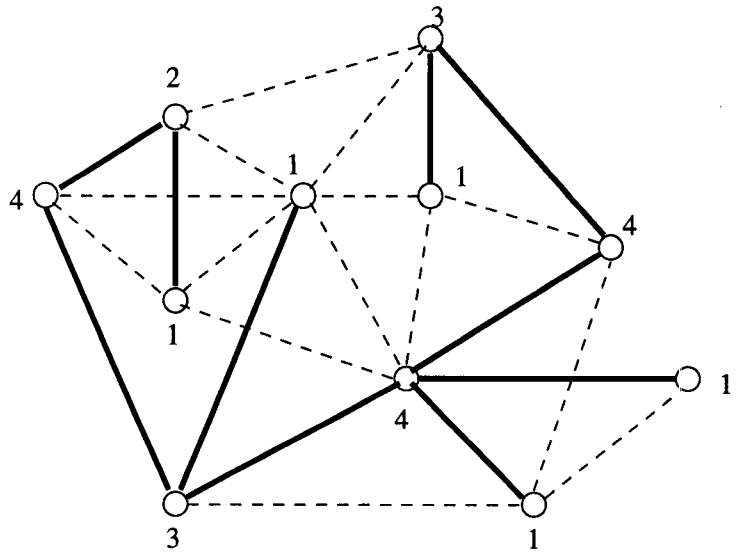


Figure 3.4: The edges of a final f-matching are in bold. Accidentally the initially bad edges are here in the matching but it is not a general rule.

$\frac{1}{53}$:rd of its capacity, in at most $c\lceil \log n \rceil$ iterations. It follows that an edge can occur in at most $2c\lceil \log n \rceil - 1$ iterations as a very good edge, being deleted from G in the last iteration. Set d to be $= 2\frac{c}{b}$. Then, the expected total number of occurrences of very good edges in the $d\lceil \log n \rceil$ iterations is greater than $e(2c\lceil \log n \rceil - 1)$, a contradiction. □

The notation "time $E\mathcal{O}(g)$" means that the expected time complexity is $\mathcal{O}(g)$.

Theorem 3.4.1 *We can compute a maximal f-matching in time $E\mathcal{O}(\log^3 n)$ on an EREW PRAM with $\mathcal{O}(n+m)$ processors.*

Proof: By Lemma 3.4.3 it is sufficient to note that a simple iteration of the block under the repeat instruction takes time $E\mathcal{O}(\log n)$ on an EREW PRAM with $\mathcal{O}(n+m)$ processors.

To implement the choose-edges stage and the next stages, we assign $\deg(v)$ processors to each vertex v. Next, we let each of the first $\lceil \frac{1}{2} f(v) \rceil$ processors to choose a random number in the range $[1, \deg(v)]$. Such a processor generates a logarithmic number of random bits in each step

and checks whether the first $\lfloor \log \deg(v) \rfloor + 1$ of them represent a number in the range $[1, \deg(v)]$. If not, the processor repeats the operation. For a sufficiently large constant c, after $c \log n$ iterations all involved processors have generated a random number in the proper range almost certainly, i.e., with probability not less than $1 - n^{-k}$ where $k > 1$. Next, for each vertex v in parallel, the random numbers generated by the processors assigned to v are sorted in logarithmic time using the $\deg(v)$ processors (in the EREW PRAM model) [9]. Further, a single processor is assigned to each element of the sorted list for v. Such a processor checks whether the key of its element is strictly greater than that of the preceding one. If so, it uses the pointer to the occurrence of the corresponding edge on the incidence list of one of its endpoint to mark the occurrence.

To perform the processor assignments in this and later stages, we respectively use parallel list ranking or prefix sums in the EREW PRAM model[36]. For example, to assign the $\deg(v)$ processors to each vertex v, we firstly compute the degrees $\deg(v)$ by parallel list ranking on the incidence lists and then the intervals of the indices of the processors to be assigned to v by parallel prefix sums. Next, each v distributes its identity to the processors with indices in its interval in logarithmic time using these processors in the EREW PRAM model. We conclude that the choose-edges stage takes time $EO(\log n)$ and $\mathcal{O}(n + m)$ processors.

To implement the bound-indegrees stage, we use the cross links between the two occurrences of the same edge on the incidence lists of its endpoints. (If they are not given in the input, we can obtain them by linking the incidence lists and then sorting the elements on the common list so the neighboring elements on the sorted list can exchange their original addresses. It takes logarithmic time and $\mathcal{O}(n + m)$ processors in the EREW PRAM model [9].) For each v in parallel, we assign the $\deg(v)$ processors of v to the consecutive edges on the incidence lists of v. Each of the processors specially marks the other occurrence of its edge by using the aforementioned links whenever its edge is marked. This yields the representation of the sets $S(v)$. Now, by using the ranks on the incidence lists computed in the previous stage, each processor assigned to a specially marked edge on an incidence list can decide whether the edge should be selected. If so, it preserves the marking of the edge. Otherwise it removes all its markings. Finally, all processors assigned to marked edge occurrences, mark the complementary occurrences by

using the cross links. In this way, we obtain a representation of the set F in logarithmic time, using $\mathcal{O}(n+m)$ processors in the EREW PRAM model.

In the repair stage, for each vertex v in parallel, we can count the number of edges in F incident to v by optimal parallel weighted list ranking of the incidence list of v. It takes logarithmic time in the EREW PRAM model [36]. In case there are exactly two edges in F on the incidence list of v and $f(v) = 1$, a random bit is distributed among the elements of the incidence list of v. If the bit is 0 the first element of F on the list is unmarked, otherwise the second one. To remove the edges in F on the original incidence lists, i.e., to update E, we can apply the straightforward procedure for sublist computation given on p. 18 in [27]. It can be easily implemented in logarithmic time using a linear number of processors in the EREW PRAM model (by keeping the information of whether the current next element is marked at each list element).

In the cleanup stage, we may assume that $\deg_{\gamma(F)}(v)$ are known from the previous stage. The removal of all edges incident to a vertex v is straightforward. The occurrences of these edges on the incidence lists of the other endpoints of the edges are obtained by marking them using the cross-links and then list shrinkage which removes marked elements. □

It seems difficult to derive a substantially sub-$\log^3 n$-time implementation of the Algorithm 3.3 in the EREW PRAM model. In the implementation of a single iteration of the block, both the random choice of $\frac{1}{2}f(v)$ edges among $\deg(v)$ edges incident to v and the updating of $f(v)$ can take logarithmic in $\deg(v)$ and $f(v)$ time respectively, independent of the number of processors assigned to v. Using the power of CRCW we could randomly choose "about" $f(v)$ edges by partitioning the set of edges into f disjoint groups of roughly the same size and then randomly choose one element from each group using the constant time random selection from [33]. However it is not clear whether the updates of vertex capacities could also be done faster using the power of concurrent writes. To achieve a $\log n$ speed-up in the CRCW PRAM model we need to modify Algorithm 3.3 preserving its main structure. The modification is three-fold:

1. In the choose-edges phase, the marking of edges is done by active

entries of capacity tables. More precisely, with each vertex v we initially associate a linear table $T(v)$ with $f(v)$ active entries. Now, during the choose-edges phase each active entry marks an edge incident to v with probability $\frac{1}{2}$ provided that the edge is not F-marked (see 3). Note that some edges can be marked by many entries of $T(v)$.

2. In the bound-degrees phase, the selecting of edges is done by the marked edges themselves. More precisely, each edge, say (v, w) randomly picks up one of its endpoints, say v, and an entry in $T(v)$. If the entry is active then exactly one of the marked edges that picked it becomes preliminary selected with probability $\frac{1}{2}$ and the entry becomes preliminary passive if the edge is really preliminary selected. Next, each of the preliminary selected edges (v, w) similarly repeats the operation for the other endpoint w. That is, it randomly picks up an entry in $T(w)$. If the entry in $T(w)$ is active then exactly one of the preliminary selected edges that picked it becomes selected and the entry as well as the entry in the table corresponding to the other endpoint of the edge become passive. All the entries in the tables $T()$ that are not passive become active.

3. The edges in F, i.e., the selected edges that remained after the clean-up phase, become F-marked. Their deletion from G and insertion into F as well as the deletion of all edges incident to a saturated vertex from G are done only in iterations whose numbers are equal to 1 mod $\lceil \log n \rceil$. Also, the contraction of the tables $T(v)$ consisting in removing all passive entries is done only in the above phases.

Lemma 3.4.4 *The modified algorithm is partially correct, i.e., if it stops then it produces a maximal f-matching of the input graph.*

Proof: The partial correctness follows from the correctness of Algorithm 3.3, and the two following facts:

i) An edge accounted to the partial f-matching F, i.e., in particular an F-marked edge, can never be attempted to be F-marked again (see 1).

ii) The total number of edges in F and the number of F-marked edges incident to a vertex v never exceeds $f(v)$ since the number of active entries in $T(v)$ is in fact equal to the current $f(v)$, i.e., to the original $f(v)$ minus the number of edges incident to v in F or F-marked (see 2). □

Lemma 3.4.5 *The modified algorithm can be implemented in time $EO(\log^2 n)$ on a CRCW PRAM with $O(n+m)$ processors.*

Proof: To simplify the exposition let us assume first that the update phase described in modification 3 is done in every iteration and under this assumption prove that the so partially modified algorithm runs in $EO(\log^2 n)$ iterations. To see this observe that for a given vertex the expected number of incident marked good edges, and then consequently of incident selected good edges and incident selected good edges surviving the clean-up phase is at least a constant fraction of the corresponding numbers for the corresponding iteration in Algorithm 3.3. This observation is non-trivial, as the selected vertices are picked among the preliminary selected vertices. The crucial point here is that a marked edge that won a competition for an active entry becomes preliminary selected with probability $\frac{1}{2}$. In this way we leave about half of the active entries free for the second round of selection. Now it is enough to decrease the lower bound on the decrease of capacity of a good vertex to its fraction in the definition of very good edges in the proof of Lemma 3.4.2, to be able to show analogously that the number of edges in the graph halves after $O(\log n)$ iterations. We conclude that the so partially modified algorithm needs $O(\log^2 n)$ iterations of the block to terminate. Now let us consider the fully modified algorithm, with the updates only in every $\lceil \log n \rceil$-th iteration and argue that still $O(\log^2 n)$ iterations are needed to terminate. The main problem here is that during the $\lceil \log n \rceil$ iterations a large part of edges incident to a vertex v can become F-marked and/or a large part of the table $T(v)$ can become passive. The former can substantially decrease the chances of an active entry to mark a non F-marked edge in the edge-choose phase compared with the situation where the marked edges are deleted. The latter can substantially decrease the chances of marked incident edges for finding an active entry in $T(v)$ in the bound-degrees phases compared with the situation

where $T(v)$ is shrinked to contain only active entries. However, the situation where at least $\frac{1}{2}f(v)$ edges incident to v are F-marked, or equivalently when $T(v)$ contains at least half of passive entries, is also good as it means that the capacity of v has been already reduced by at least half. On the other hand, if no more than $\frac{1}{2}f(v)$ edges are F-marked, or equivalently if no more than half of the entries in $T(v)$ are passive, the mentioned chances of active entries or marked edges incident to v respectively drop only by half. Therefore, the expected number of iterations increases only by a constant factor.
It remains to estimate the parallel complexity of a single iteration of the block in the modified algorithm. Using the constant time random selection from [33], all the random choices in the choose-edge, bound-indegrees and clean-up phases can be done in constant time on a CRCW PRAM with $\mathcal{O}(n+m)$ processors. Also the choices of one among several edges competing for the same entry in a table $T()$ can be done in constant time on an arbitrary CRCW PRAM. We conclude that a single iteration whose number is not 1 modulo $\lceil \log n \rceil$ takes constant time on an arbitrary CRCW PRAM with $\mathcal{O}(n+m)$ processors. The whole updating occurring in the remaining iterations can be easily done in time $\mathcal{O}(\log n)$ on an arbitrary CRCW PRAM. For instance, to shrink the tables $T()$ we can use list ranking. As the number of iterations where the updating occurs is $\mathcal{O}(\log n)$ we conclude that the whole modified algorithm runs in time $\mathcal{O}(\log^2 n)$ on an arbitrary CRCW PRAM with $\mathcal{O}(n+m)$ processors. □

Theorem 3.4.2 *Let G a graph with a positive integer capacity function f defined on its vertices. A maximal f-matching can be found in time $E\mathcal{O}(\log^2 n)$ on an arbitrary CRCW PRAM with $\mathcal{O}(n+m)$ processors.*

Chapter 4

Hypergraphs

Since most concepts of science are relatively simple once you understand them, any ambitious scientist must, in self-preservation, prevent his colleagues from discovering that his ideas are simple too.

Nicolas Vanserg.

4.1 Introduction

A hypergraph is a natural generalization of a graph.
Let $V = \{v_1, v_2, \ldots, v_n\}$ be a finite set, and let $E = \{E_1, E_2, \ldots, E_m\}$ be a family of non-empty subsets of V such that:

$$\bigcup_{1 \leq i \leq m} E_i = V$$

The couple $H = (V, E)$ is called a *hypergraph*. The elements $v_1, v_2, \ldots, v_n$ are called the *vertices* and the sets $E_1, E_2, \ldots, E_m$ are called the *edges*. For a node $v \in V$ the *degree* $\deg(v)$ of v in H is the number of edges in E it belongs to. The maximum degree of a node in V is called the *valence* of H, and the maximum cardinality of an edge in E is called the *dimension* of H. If the edges of the hypergraph are all distinct, the hypergraph is said to be *simple*. A simple hypergraph of dimension two is simply an undirected graph (if singletons are neglected). In this chapter, all hypergraphs are simple.

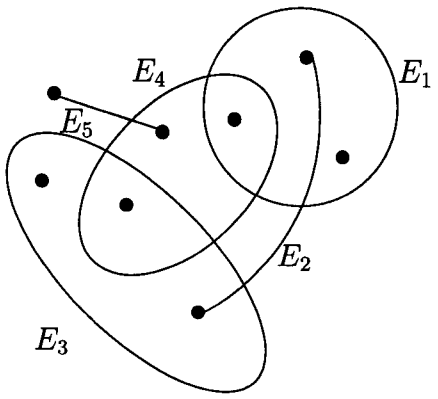


Figure 4.1: Example of a hypergraph $= (V; E_1, E_2, E_3, E_4, E_5)$.

A hypergraph is shown in Fig 4.1. An edge E_i with $|E_i| > 2$, is drawn as a curve encircling all the vertices in E_i. An edge with $|E_i| = 2$ is drawn as a curve connecting its two vertices.

An independent set of H is a subset of V which doesn't include any edge in E. A maximal independent set (MIS, for short) of H is an independent set which is not a subset of any other independent set of H.

For a hypergraph $H = (V, E)$, let G_H denote the corresponding bipartite graph $(V \cup E, E')$ where $(v, e) \in E'$ iff $v \in e$. It is easy to see that:

Theorem 4.1.1 *A subset W of V is independent in H iff $W \cup E$ is f-dependent in G_H where for $v \in V$, $f(v) = \deg(v)$ and for $e \in E$, $f(e) = |e| - 1$.*

Thus, the problem of finding a maximal independent set in a hypergraph is a special case of the problem of finding a maximal f-dependent set in a graph (See Section 2.4).

4.2 MIS in hypergraphs

A MIS of a hypergraph can be trivially computed in polynomial time by a greedy method. The parallel complexity status of finding a MIS

of an arbitrary hypergraph is regarded as a major open problem in parallel complexity theory [36]. When no restrictions on dimension are assumed the only non-trivial upper bound follows from a parallel randomized search method due to Karp, Upfal and Wigderson [38]. Their method yields $\mathcal{O}(\sqrt{|V|}\log(|V|+|E|))$ expected-time bound on finding a MIS of a hypergraph in the EREW PRAM model $H=(V,E)$ (see [40]). For hypergraphs of constant dimension, Kelsen has recently proved a parallel randomized algorithm due to Beame and Luby to run in polylogarithmic time using a linear number of processors [40]. Thus, the so restricted problem is in RNC. By derandomizing the aforementioned algorithm Kelsen has also shown that a MIS for a hypergraph of constant dimension can be found in time n^{ϵ}, for any given $\epsilon > 0$. For hypergraphs of dimension 2, i.e., for graphs, several NC algorithms for MIS are known [28, 39, 45]. The first of them is due to Karp and Wigderson [39], the simplest is due to Luby [45], and the most efficient is due to Goldberg and Spencer [28]. The latter algorithm has been recently generalized to include hypergraphs of dimension 3 independently by Dahlhaus, Karpinski and Kelsen [14]. In this way, the membership of the MIS problem for hypergraphs of dimension 3 in NC has been established. It seems that the dimension 3 is a limit for the method due to Goldberg and Spencer [14]. Also, Kelsen [40] admits that a new approach has to be found in order to derive efficient parallel algorithms for MIS in hypergraphs of non-constant dimension.

4.3 MIS in sparse hypergraphs

In this section we consider hypergraphs of arbitrary dimension that are hereditary sparse. To formalize the sparsity property we extend the known concept of graph *arboricity* to include hypergraphs. Recall that the arboricity $\Upsilon(G)$ of a graph G is the minimum number of forests the edges of G can be partitioned into. For example, graphs of bounded genus and partial k-trees have constant arboricity. Analogously, we define the arboricity $\Upsilon(H)$ of a hypergraph H as the minimum number of acyclic hypergraphs the edges of H can be divided into.

We show that a maximal independent set in a hypergraph H on n nodes can be found in time $\mathcal{O}(\Upsilon(H)^2\log^4 n)$ on a CREW PRAM with $\mathcal{O}(n\Upsilon(H)(\frac{\Upsilon(H)}{\log n}+1))$ processors, or in time $\mathcal{O}(\Upsilon(H)^6\log^2 n)$ on a CREW

PRAM with $\mathcal{O}(n\Upsilon(H)^2)$ processors. Thus, if H is of poly-logarithmic arboricity, i.e., $\Upsilon(H) = \mathcal{O}(\log^k n)$ for some integer constant k, then a maximal independent set in H can be found in poly-logarithmic time using a polynomial number of processors.

Also, if H is of constant arboricity, i.e., $\Upsilon(H) = \mathcal{O}(1)$, then a maximal independent set in H can be constructed in time $\mathcal{O}(\log^2 n)$ on a CREW PRAM with $\mathcal{O}(n)$ processors.

4.3.1 Hypergraph arboricity

A hypergraph is acyclic if it doesn't contain any chain of edges such that any two consecutive edges and the first and the last edge in the chain respectively overlap [5]. The following characterization of acyclic hypergraphs will be useful (Proposition 4 p. 392 in [5]).

Lemma 4.3.1 *A hypergraph $H = (V, E)$ with n nodes and p connected components is acyclic iff*

$$\sum_{e \in E} (|e| - 1) = n - p.$$

For simplicity, we shall denote the set of restrictions of edges in a set E to a node subset U, i.e., $\{e \cap U \mid e \in E\}$, by $E \cap U$. Also, we shall say that a hypergraph $F = (U, D)$ is a sub-hypergraph of a hypergraph $H = (V, E)$ if $U \subset V$, and $D \subset E \cap U$. It follows easily that for any sub-hypergraph F of a hypergraph H, the inequality $\Upsilon(F) \leq \Upsilon(H)$ holds. Hence, we obtain the following theorem by Lemma 4.3.1.

Theorem 4.3.1 *For any sub-hypergraph $F = (U, D)$ of a hypergraph H the inequality $\sum_{e \in D}(|e| - 1) < \Upsilon(H) \times |U|$ holds. In particular, the number of non-singleton edges of F is smaller than $\Upsilon(H) \times |U|$.*

Thus, the notion of hypergraph or graph arboricity corresponds to the notion of hereditary or inherent sparsity.

By the *size of a hypergraph* we shall mean the sum of cardinalities of its edges and the number of its vertices.

Corollary 4.3.2 *For any hypergraph H on n vertices, the size of H is smaller than $2n(\Upsilon(H) + 1)$.*

4.3.2 MIS in hypergraphs of bounded arboricity

Our parallel algorithm for finding a maximal independent set in a hypergraph of bounded arboricity can be seen as an NC Turing-like reduction of the original problem to the corresponding problem for a hypergraph with bounded dimension and bounded valence. Its subroutine for finding a maximum independent set in a hypergraph (V, E) with dimension and valence bounded by b is denoted by $MIS_B(V, E, b)$ (ALGORITHM 4.2).

Algorithm $MIS_A(H)$
input : A hypergraph $H = (V, E)$ on n nodes where $\Upsilon(H) \leq d$.
output : A maximal independent set M in H.
method :
if $|V| \leq 1$ then **output** $\emptyset$ and stop;
$D \leftarrow \emptyset$;
for each e satisfying $|e| > 5d$ insert a node in e into D;
insert each node occuring in $> 5d$ edges in E into D;
$C \leftarrow V \setminus D$;
$E_0 \leftarrow \{e \in E \mid e \cap D = \emptyset\}$;
$M \leftarrow MIS_B(C, E_0, 5d)$;
$F \leftarrow \{e \in E \setminus E_0 \mid e \cap (C \setminus M) = \emptyset\}$;
output $M \cup MIS_A(D, F \cap D)$;
end MIS_A

ALGORITHM 4.1

Lemma 4.3.3 ALGORITHM 4.1 *MIS_A is partially correct, i.e., if it stops then the set Q to output is a maximal independent set in H.*

Proof: Since C doesn't contain nodes of degree $> 5d$ and E_0 is free from edges of size $> 5d$, the subroutine $MIS_B(C, E_0, 5d)$ correctly returns a MIS M in the hypergraph (C, E_0). By the maximality of M, for any node v in $C \setminus M$, there is an edge $e \in E_0$ such that $v \in e$ and $e \setminus M = \{v\}$. Hence, no node in $C \setminus M$ can occur in any independent set in H including M. Therefore, it remains only to extend M by a maximal subset of D. Since $C \cap D = \emptyset$, the edges in E_0 as well as the edges in $E \setminus E_0$ with a non-empty intersection with $C \setminus M$ never can be completely filled with nodes in $D \cup M$. Therefore, we can disregard them constructing such a D-extension. Hence, it is both sufficient and necessary to extend M by a maximal subset of D so that the edges consisting solely of nodes in M and D won't be completely filled, i.e., by a MIS in the hypergraph

$(D, F \cap D)$. The hypergraph $(D, F \cap D)$ has arboricity $\leq d$. Hence, we may assume inductively that $MIS_A(D, F \cap D)$ correctly outputs a MIS in $(D, F \cap D)$ since $|D| < |V|$ holds by the following lemma. □

Lemma 4.3.4 *Let $n = |V|$, where $H = (V, E)$ is the input hypergraph in MIS_A. The inequality $|D| < \frac{4}{5}n$ holds. Hence, the recursion depth of the algorithm MIS_A is $\mathcal{O}(\log n)$.*

Proof: Consider the input hypergraph $H = (V, E)$ where $n = |V|$. Since $\Upsilon(H) \leq d$, the number m of edges in E where $|e| > 5d$ satisfies $5dm \leq dn$ by Theorem 4.3.1. Also, the number l of nodes in H of degree $> 5d$ satisfies $5dl \leq \sum_{e \in E}(|e| - 1) + |E|$ which yields $5dl \leq 2dn + n$ by Theorem 4.3.1. We conclude that $|D| < \frac{4}{5}n$. □

Lemma 4.3.5 *Suppose that a maximal independent set in a hypergraph on n nodes with dimension and valence bounded by k can be found in time $T_h(n, k)$ using a PRAM with $P_h(n, k)$ processors. Algorithm MIS_A can be implemented in time $\mathcal{O}(\log n\ (T_h(n, 5d) + \log(nd)))$ on an CRCW PRAM with $\mathcal{O}(\frac{nd}{\log(nd)}) + P_h(n, 5d)$ processors.*

Proof: By Theorem 4.3.1, the total size of H, i.e., the sum of cardinalities of its edges is $\mathcal{O}(nd)$.

To implement the set operations quickly using a linear number of processors, we represent the node sets different from edges with n element vectors, each of them with 1 on the i:th position if and only if the i:th node in H is currently in the set. Analogously, we represent the sets of edges with $\mathcal{O}(nd)$ element vectors. On the contrary, an edge of H is encoded in a table listing the numbers of the contained nodes.

The edge tables as well as the edge dimensions and the node degrees can be easily determined in logarithmic time within the processor bounds given in the lemma by parallel list ranking and parallel merge sort [36, 9].

By the standard simulation of a CRCW PRAM on an EREW PRAM [36], we can insert for each e satisfying $|e| > 5d$ a node in e into D, and to compute E_0 and F, both in time $\mathcal{O}(\log(nd))$ on an EREW PRAM with $\mathcal{O}(nd)$ processors.

Hence, all the instructions different from the calls of MIS_B and MIS_A can be implemented in time $\mathcal{O}(\log(nd))$ on an EREW PRAM with $\mathcal{O}(nd)$ processors.

A maximal independent set in the hypergraph (C, E_0) can be found in time $T_h(n, 5d)$ using $P_h(n, 5d)$ CREW PRAM processors.

We conclude that all instructions but for the recursive call of MIS_A totally take time $T_h(n, 5d) + \mathcal{O}(\log(nd))$ on a CREW PRAM with $\mathcal{O}(nd) + P_h(n, 5d)$ processors. This combined with Lemma 4.3.4 and $|D| < n$ yields the thesis. □

4.3.3 MIS in hypergraphs of bounded dimension and valence

In this section, we present a parallel algorithm for finding a maximal independent set in a hypergraph of bounded dimension and valence which is a specialization of the parallel method for maximal f-dependent set in a graph for bounded f from section 2.4. Due to the specialization more precise bounds are derived.

Algorithm $MIS_B(V, E, b)$
input : A hypergraph $H = (V, E)$ with dimension and valence bounded by b.
output : A maximal independent set K in H.
method :

$K \leftarrow \emptyset$;
$B \leftarrow V \setminus \bigcup_{e \in E \& |e|=1} e$;
while $B \neq \emptyset$ **do**
  $G \leftarrow$ the graph whose set of nodes is B
    such that (v, w) is an edge of G iff
    v and w belong to the same edge in E;
  $L \leftarrow$ a maximal independent set in G;
  $K \leftarrow K \cup L$;
  $S \leftarrow \{e \in E \mid |(e \cap K)| = |e| - 1\}$;
  $B \leftarrow B \setminus (L \cup \bigcup_{e \in S} e)$;
endwhile
output K;
end MIS_B

ALGORITHM 4.2

Lemma 4.3.6 *The block under the while statement is iterated* $\mathcal{O}(b^2)$ *times.*

Proof: Each iteration after which a node v remains in B results in inserting at least one node in at least one of the at most b edges containing v into K. Therefore, after $\leq b(b-2)+1$ iterations at least one of the edges will be included into S and consequently v will disappear from B for good. □

Lemma 4.3.7 *Algorithm MIS_B is correct.*

Proof: The augmentation of K by L is correct since no node in L is contained in an edge e having already $|e|-1$ nodes in K, and L is also an independent set in H. This combined with Lemma 4.3.6 yields the total correctness □

Lemma 4.3.8 *Suppose that a maximal independent set in a graph on n nodes with valence bounded by k can be found in time $T_g(n,k)$ on a CREW PRAM with $P_g(n,k)$ processors. Algorithm MIS_B can be implemented in time $\mathcal{O}(b^2\log(nb)+b^2T_g(n,b^2))$ on a CREW PRAM with $\mathcal{O}(nb(1+\frac{b}{\log(nb)}))+P_g(n,b^2)$ processors.*

Proof: By Lemma 4.3.6, we can replace the while statement by a "for" loop with the number of iterations $\mathcal{O}(b^2)$ to avoid the test for emptiness of B. Also, the set instructions can be implemented in constant time, using $\mathcal{O}(nb)$ CREW PRAM processors, analogously as in the proof of Lemma 4.3.5.

To construct the auxiliary graph G, we proceed as follows. For each node v of H, we form the list $E(v)$ of the at most b edges containing v. Using the input representation of H as a sequence of node lists representing the edges of H, it can be done by sorting in $\mathcal{O}(\log(nb))$ time on an EREW PRAM with $\mathcal{O}(nb)$ processors [9]. Further, using parallel list ranking, we assign $\mathcal{O}(b^2)$ processors to consecutive groups of $\mathcal{O}(\log(nb))$ elements on such a list $E(v)$ to list out all nodes adjacent to v in G in time $\mathcal{O}(\log(nb))$.

The valence of G is $\leq b^2$. Hence, a maximal independent set in G can be found in time $T_g(n,b^2)$ on a CREW PRAM with $P_g(n,b^2)$ processors by our assumptions. □

Goldberg and Spencer have proved that a maximal independent set in a graph on n nodes and m edges can be computed in time $\mathcal{O}(\log^3 n)$ on

a EREW PRAM with $\mathcal{O}(\frac{n+m}{\log n})$ processors [29]. Further, Goldberg and Plotkin have shown that if the input graph has valence bounded by k then its maximal independent set can be constructed in time $O(k^2 \log n)$ on an EREW PRAM with $\mathcal{O}(n+m))$ processors [30]. Combining these facts with Lemmata 4.3.3, 4.3.5, 4.3.7, 4.3.8, we obtain our main result:

Theorem 4.3.2 *A maximal independent set in a hypergraph H on n nodes can be found:*

1. *in time $\mathcal{O}(\Upsilon(H)^2 \log^4 n)$ on a CREW PRAM with $\mathcal{O}(n\Upsilon(H)(1 + \frac{\Upsilon(H)}{\log n}))$ processors;*
2. *in time $\mathcal{O}(\Upsilon(H)^6 \log^2 n)$ on a CREW PRAM with $\mathcal{O}(n\Upsilon(H)^2)$ processors.*

Corollary 4.3.9 *A maximal independent set in a hypergraph H of $\mathcal{O}(1)$ arboricity can be found in time $\mathcal{O}(\log^2 n)$ on a CREW PRAM with $\mathcal{O}(n)$ processors.*

Chapter 5

Conclusions

All human knowledge begins with intuitions, proceeds to concepts, and ends in ideas

Immanuel Kant

We have studied the parallel complexity of a family of fundamental combinatorial problems which have a numerous applications in different areas.

Maximum k-dependent set

The computation of a maximum k-dependent set has been shown to be probably unfeasible problem. Recent results show that even for bipartite graphs and $k \geq 1$ the problem remains NP-complete [15]. So further research should be centered on finding algorithms for restricted graph families, or studying the approximability of the problem.

For a non-negative k and a graph G, let $D_k(G)$ denote the maximum cardinality of a k-dependent set in G. Clearly, we have $D_0(G) \leq D_1(G) \leq D_2(G) \ldots$. For a non-negative integer b, a b-matching in a graph G is a subset of the set of edges of G such that for each vertex at most b edges incident to the vertex are in the subset; in other words, an f-matching where $f(v) = b$ for all vertices v. Let $M_b(G)$ stand for the maximum cardinality of a b-matching in G. It would be interesting to find non-trivial relations between $D_k(G)$'s and $M_b(G)$'s for b's related to k. For instance it is known that $M_1(G) \leq D_0(G)$, and for bipartite

graphs $M_1(G) = D_0(G)$.

Maximal f-dependent sets and MIS on hypergraphs

For non-dense graphs our NC algorithm for computing maximal k-dependent sets is far from being optimal in the sense of the time-processor product. It seems that more processor-efficient NC algorithms for maximal k-dependent set can be derived in the special cases of $k = 1, 2$ and for planar graphs (see [22]). However, the ultimate goal here would be to derive an NC algorithm for maximal k-dependent set in the general case such that the time-processor product would be within a logarithmic factor from the size of the input graph. We would like to design faster parallel algorithms for different graph families. The generalization of our algorithm for maximal k-dependent set to include maximal f-dependent set (see Theorem 2.4.1) runs in poly-logarithmic time only if the maximum value of f is poly-logarithmic in the input size. Thus, the problem of whether one can construct a maximal f-dependent set in the general case of f using an NC algorithm is also open. Surprisingly we have found the latter problem to be essentially equivalent to the problem of whether a maximal independent set of a hypergraph can be constructed by an NC algorithm (when hyper-edge size is constantly bounded a randomized NC solution is known). For the hypergraph problem we have provided an NC solution in the bounded arboricity case.

Maximal f-matching

Our parallel algorithms for maximal f-matching are not optimal. The deterministic algorithm has the same complexity bounds as the best known maximal 1-matching algorithm, due to Israeli and Shiloachs [34]. The work of our randomized algorithm is a logarithmic factor from the best known randomized parallel algorithm for maximal 1-matching (Israeli and Itai [33]). We have presented a interesting method of achieving a logarithmic speed-up in the CRCW model for our EREW algorithm.

We believe that it is possible to improve the bounds of our algorithms and that it is possible to develop optimal parallel algorithms for this problem. The case of co-graphs is also interesting as there are known optimal parallel algorithms for maximum 1-matchings in these graphs.

Bibliography

[1] S. ARNBORG, J. LAGERGREN AND D. SEESE, *Problems Easy for Tree Decomposable Graphs.* In Proc. Inter. Colloq. on Automata, Languages and Programming, Tampere 1988. Lecture Notes in Computer Science 317, Springer Verlag, pp.38-133 .

[2] T. ASANO, *Graphical Degree Sequence Problems with Connectivity Requirements.* Proc. ISAAC'93, Hong Kong, Springer, LNCS 762, pp. 88-97.

[3] B. AWERBUCH, A. ISRAELI AND Y. SHILOACH, *Finding Euler circuits in logarithmic parallel time*, Proc. 16th Ann. ACM Symp. on Theory of Computing (1984), pp.249-257.

[4] B. S. BAKER, *Approximation algorithms for NP-complete problems on planar graphs.* In Proc. 24th Ann, Symp. on Found. of Comp. Sci.(1983) pp. 265-273.

[5] C. BERGE, *Graphs and Hypergraphs*, (North-Holland Mathematical Library, 1973)

[6] P. BERMAN AND T. FUJITO *On Approximation Properties of the Independent Set Problem for Degree 3 Graphs* In Proc. 4th Int. Workshop on Algorithms and Data-structures, WADS'95, Lecture Notes in Computer Science 955, Springer Verlag, pp. 449-460.

[7] S. CARLSSON, Y. IGARASHI, K. KANAI, A. LINGAS, K. MIURA AND OLA PETERSSON, *Information Disseminating Schemes for Fault Tolerance in Hypercubes.* IEICE Trans. Fundamentals. Vol.E75-A. No 2 February 1992, pp. 255-260.

[8] E. COHEN, Approximate max flow on small depth networks. Proc. 33rd FOCS, 1992, pp. 648–658.

[9] R. COLE, *Parallel merge sort.* SIAM J. Comput., vol 17, No. 4, 1988, pp 770-785

[10] R. COLE AND U. VISHKIN, *Optimal parallel algorithms for expression tree evaluation and list ranking.* In VLSI Algorithms and architectures. 3rd Aegean Workshop on Computing, AWOC 88 pp 91-100.

[11] S. COOK, *The Classification of Problems which have Fast Parallel Algorithms*, Proceedings of the 1983 International FCT-Conference, Borgholm, Sweden, Lecture Notes in Computer Science (1983) 78-93.

[12] S. COOK, *The taxonomy of problems with fast parallel algorithms*, In Information and Control, Vol 64, Nos. 1-3, 1985, Academic Press, New York.

[13] E. DAHLHAUS AND M. KARPINSKI, *A Fast Parallel Algorithm for Computing All Maximal Cliques in a Graph and the Related Problems.* In the Proceeding of the 1st Scandinavian Workshop on Algorithm Theory, SWAT 88, Lecture Notes in Computer Science 318, Springer Verlag, pp. 139-144.

[14] E. DAHLHAUS, M. KARPINSKI AND P. KELSEN, *An Efficient Parallel Algorithm for Computing a Maximal Independent Set in a Hypergraph of Dimension 3*, to appear in Information Processing Letters.

[15] A. DESSMARK, K. JANSEN AND A. LINGAS, *The maximum k-dependent and f-dependent set problem* Proceedings of the 4th Annual International Symposium on Algorithms and Computation ISAC'93, Lecture Notes in Computer Science 762, Springer Verlag, pp 88-97.

[16] K. DIKS, O. GARRIDO AND A. LINGAS, *Parallel algorithms for finding maximal k-dependent sets and maximal f-matchings*, International Journal of Foundations of Computer Science, Vol 4, No 2, pp. 179-192.

[17] H. DJIDJEV, O. GARRIDO, C. LEVCOPOULOS AND A. LINGAS, *On the maximum q-dependent set problem.* International Conference for Young Computer Scientists 91 ICYCS91. pp.271-274.

[18] P. ERDÖS AND T. GALLAI, *Graphs with prescribed degrees,* In Mat. Lapok 11, 1960, pp.264-274.

[19] Z. GALIL AND V. PAN. *Improved processor bounds for combinatorial problems in RNC.* Combinatorica, 8, 1988, pp. 189-200.

[20] M. R. GAREY, D. S. JOHNSON, *Computers and Intractability* (W. H. Freeman and Company, San Francisco, 1979).

[21] O. GARRIDO, S. JAROMINEK, A.LINGAS AND W. RYTTER, *A Simple Randomized Parallel Algorithm for Maximal f-Matchings,* In the Proc of the 1st. Latin American Theoretical Informatics LATIN'92.Lecture Notes in Computer Science 589, Springer Verlag. pp.165-176. To appear in Information Processing Letters.

[22] O. GARRIDO, *The Complexity of the q-dependent set problem,* Thesis for the degree of Masters of Science, Dept. Computer Science, Lund University.

[23] O. GARRIDO, *On the Realization of Degree Sequences in Parallel,* Internal Report, LU-CS-TR:93-119, Dept. Computer Science, Lund University.

[24] O. GARRIDO, P. KELSEN AND A. LINGAS, *A simple NC-algorithm for a maximal independent set in a hypergraph of poly-log arboricity* To appear in Information Processing Letters.

[25] O. GARRIDO AND A. LINGAS, *An NC-algorithm for a maximal independent set in a hypergraph of poly-log arboricity* In Proc. XV International Conference of the C.C.S.S., 1995.

[26] H. GAZIT AND G. L. MILLER *A parallel Algorithm for Finding a Separator in Planar Graphs.* In Proc. 28th Symp. on Foundations of Computer Science, 1987.

[27] A. GIBBONS AND W. RYTTER, *Efficient Parallel Algorithms* (Cambridge University Press, Cambridge, 1988).

[28] M. GOLDBERG AND T. SPENCER, *A New Parallel Algorithm for the Maximal Independent Set Problem.* In Proc. 28th Symp. on Foundations of Computer Science, 1987.

[29] M. GOLDBERG AND T. SPENCER, *Constructing a Maximal Independent Set in Parallel* SIAM, J. Disc. Math, Vol 2, No 3 (1989), pp.322-328.

[30] A. V. GOLDBERG AND S. A. PLOTKIN, *Parallel* $(\Delta + 1)$- *Coloring of Constant-degree Graphs.* Information Processing Letters 25 (1987) pp. 241-245.

[31] T. HAGERUP AND C. RÜB, *A guided tour of Chernoff bounds.* Information Processing Letters 33 (1989/90) 305-308.

[32] S.L. HAKIMI, *On the realizability of a set of integers as degrees of the vertices of a linear graph*, J. SIAM 10, 1962, pp. 496-506.

[33] A. ISRAELI AND A. ITAI, *A fast and simple randomized parallel algorithm for maximal matching.* Information Processing Letters 22 (1986) 77-80.

[34] A. ISRAELI AND Y. SHILOACH, *An improved parallel algorithm for maximal matching.* Information Processing Letters 22 (1986) pp. 57-60.

[35] H.J. KARLOFF, *A Las Vegas RNC algorithm for maximum matching.* Combinatorica 6(4), pp. 387:391, 1986.

[36] R. M. KARP AND V. RAMACHANDRAN, *A Survey of Parallel Algorithms for Shared-Memory Machines.* Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity, J. van Leeuwen, Editor, Elsevier Science Publisher, B.V. 1990, ISBN 0444 88071 2

[37] R. M. KARP, E. UPFAL AND A. WIGDERSON, *Constructing a perfect matching is in random NC*, Combinatorica 6, 1986 pp. 35-48.

[38] R. M. KARP, E. UPFAL AND A. WIGDERSON, *The complexity of parallel search*, JCSS vol. 36, 1988, pp. 225-253.

[39] R. M. KARP AND A. WIGDERSON, *A Fast Parallel Algorithm for the Maximal Independent Set Problem.* In Proceedings of the 16th Annual ACM Symposium on Theory of Computing, 1984.

[40] P. KELSEN, *On the Parallel Complexity of Computing a Maximal Independent Set in a Hypergraph*, Proc. of the 24th Annual ACM Symposium on Theory of Computing, 1992.

[41] R.E. LADNER AND M.J. FISCHER, *Parallel Prefix Computations.* In J. ACM. 27, 1980, pp.831-838.

[42] C. LEVCOPOULOS, A. LINGAS, O. PETERSSON AND W. RYTTER, *Optimal parallel algorithms for testing isomorphism of trees and outerplanar graphs.* Proceedings to 10th FST-TCS, Bengalore, India, Lecture Notes in Computer Science 472, 1990, pp. 204-214.

[43] R. J. LIPTON AND R. E. TARJAN *Applications of planar separator theorem.* In SIAM J. Comput. 9, 3 (1980) pp. 615-627.

[44] L. LOVÁSZ AND M. D. PLUMMER, *Matching Theory, Annals of Discrete Mathematics* (29). North-Holland Mathematics Studies 121. Elsevier Science Publishers B. V. ISBN 0444 879161.

[45] M. LUBY *A simple parallel algorithm for the maximal independent set problem.* In SIAM J.Comput. 15, 4 (1986) pp. 1036-1053.

[46] A. MAHESHWARI, Personal communication.

[47] E. W. MAYR AND A. S. SUBRAMANIAN *The complexity of circuit value and network stability.* In Proc. Structure in Complexity Theory (4th Ann. IEEE Conf.)(1989) pp. 114-123.

[48] S. MICALI AND V.V. VAZIRANI *An $\mathcal{O}(\sqrt{V}\cdot E)$ algorithm for finding maximum matching in general graphs*, In the Proc. 21th. Ann. IEEE Symp. Foundations of Computer Science, Syracuse 1980, pp. 17-27

[49] K. MULMULEY, U.V. VAZIRANI, AND V.V. VAZIRANI, *Matching is as easy as matrix inversion.* Combinatorica 7(1), pp. 105-113.

[50] C. N. K. OSIAKWAN AND S. G. AKL, *Optimal parallel algorithms for b-matchings in trees.* Proceedings to Optimal Algorithms International Symposium Lecture notes in Computer Science 401 (1989) 274-308.

[51] N. J. PIPPENGER, *On simultaneous resource bounds.* In the Proc. 20th. Annual Symp. on Foundation of Computer Science, 1979, pp 307-311.

[52] Y. SHILOACH AND U. VISHKIN, *An $\mathcal{O}(\log n)$ parallel connectivity algorithm*, J. Algorithms 3 (1982), pp. 56-67.

[53] W.T. TUTTE *A Short Proof of the Factor Theorem for Finite Graphs*, Canad. J. Math. 6, 1954, pp. 347-352.