# A new $\mathcal{NC}$ Algorithm for Perfect Matching in Bipartite Cubic Graphs

Roded Sharan Avi Wigderson *

Institute of Computer Science, Hebrew University of Jerusalem, Israel.
E-mail: roded@math.tau.ac.il, avi@cs.huji.ac.il

## Abstract

*The purpose of this paper is to introduce a new approach to the problem of computing perfect matchings in fast deterministic parallel time. In particular, this approach yields a new algorithm which finds a perfect matching in bipartite cubic graphs in time $O(\log^2 n)$ and $O(n\alpha(n)/\log n)$ processors in the arbitrary CRCW PRAM model.*

## 1. Introduction

Given a graph $G = (V, E)$ with $n$ vertices and $m$ edges, a *Matching* is a set of edges $M \subset E$, such that no two edges of $M$ are adjacent in $G$. A *Maximum Matching* is a matching of maximum cardinality in $G$ and a *Perfect Matching* is a matching which is incident to all vertices of $G$. The *Perfect Matching Decision Problem* is to determine if $G$ has a perfect matching. The *Search Problem* is to exhibit a perfect matching, if such exists. The *Maximum Matching Problem* is to find a matching of maximum cardinality in $G$.

The first sequential polynomial algorithm for the maximum matching problem was given by Edmonds [4], requiring $O(n^4)$ time. This algorithm as many others following it, solves the problem by finding a partial matching and augmenting it to a maximum matching. The best algorithms currently known for this problem, by Micali and Vazirani [13, 20] and by Blum [1] work in $O(m\sqrt{n})$ time.

The methods used to solve the maximum matching problem for the sequential case, do not seem to parallelize. Even the problem of finding a single augmenting path is not known to be in $\mathcal{NC}$. New techniques, using tools from linear algebra, were developed to cope with the problem. All these methods are based on a theorem by Tutte [18] which reduces the perfect matching decision problem to deciding if a certain Tutte-matrix of the input graph is non-singular. This also leads to an $\mathcal{RNC}$ algorithm for solving the decision problem (see [11]).

*This work was supported by USA-Israel BSF grant 92-00106 and by a Wolfeson research award administered by the Israeli Academy of Sciences.

The first parallel $\mathcal{RNC}$ algorithm for constructing a maximum matching was given by Karp, Upfal and Wigderson [9]. Another algorithm was later given by Mulmuley, Vazirani and Vazirani [15], and a Las-Vegas extension of both algorithms was given by Karloff [7].

Deterministic $\mathcal{NC}$ algorithms, even for the decision problem, are not known, although special cases of the perfect matching search problem turned out to be in $\mathcal{NC}$. Vazirani [19] gave an $\mathcal{NC}$ algorithm to count the number of perfect matchings in $K_{3,3}$-free graphs. Grigoriev and Karpinski [5] solved the search problem for the case that the graph has only polynomially many perfect matchings, and Lev, Pippenger and Valiant [10] gave an $\mathcal{NC}$ algorithm to find a perfect matching in bipartite d-regular graphs. The last algorithm works in $O(\log^2 n \log d)$ time using $O(m)$ processors.

In this work, we present a new approach towards the perfect matching problem. This approach yields a new $\mathcal{NC}$ algorithm to solve the perfect matching search problem for bipartite cubic graphs. The algorithm works in $O(\log^2 n)$ time using $O(n\alpha(n)/\log n)$ processors in the arbitrary CRCW PRAM model, improving the processor bound in [10], which requires (for bipartite cubic graphs) $O(\log^2 n)$ time and $O(n)$ processors. The methods we use seem to be of general nature and some stages of our algorithm are shown for general graphs as well.

Let $G = (V, E)$ be the input graph, $|V| = n$. Our Algorithm starts with a *Pseudo Perfect Matching* of $G$, which is a subgraph of $G$ with every vertex having odd degree in it. We show how find one and how to efficiently augment it until a perfect matching is produced.

The augmentation is done by finding certain cycles in $G$ and by xoring them to the pseudo perfect matching, reducing the number of edges in it and increasing its number of connected components. In other words the approach we take is augmenting by decreasing the number of edges, rather than augmenting by increasing as done in most sequential algorithms for the problem.

In total $O(\log n)$ phases are carried out. The input to each phase is a pseudo perfect matching and the output of

each phase is a new pseudo perfect matching, with number of 3-degree vertices in it, reduced by a constant factor. At the end, a perfect matching is obtained.

We also show a sequential implementation of our algorithm working in $O(n \log n)$ sequential time.

## 2. Preliminaries

Let $G = (V, E)$ be an undirected simple graph.

**Definition 2.1** *A* **Pseudo Perfect Matching** *of $G$ is a subgraph $M$, with every vertex having odd degree in it.*

**Definition 2.2** *A* **3-vertex** *of a pseudo perfect matching $M$, is a vertex of degree 3 in $M$.*

**Definition 2.3** *The* **Complement** *in $G$ of a pseudo perfect matching $M$, is the graph $G - M$, where isolated vertices are excluded.*

The *Cycle Space* of $G$ is a vector space over $GF(2)$, containing all incidence vectors representing cycles in $G$, and closed with respect to the xor operation. A vector in the cycle space is called a *Cycle Vector*.

**Definition 2.4** *An* **Augmenting Cycle** *of $G$ with respect to a subgraph $H$, is a cycle vector $C$ in $G$, such that $H \oplus C$ has less edges than $H$.*

**Definition 2.5** *A* **Good Cycle** *of $G$, with respect to a pseudo perfect matching $M$, is a cycle vector $C$, such that $C - M$ is a matching.*

Dahlhaus and Karpinski show in [3] that the perfect matching problem for general graphs is $\mathcal{NC}$-equivalent to the perfect matching problem restricted to graphs of maximum degree 3 and this reduction preserves bipartiteness.

We will adopt this reduction and from now on restrict ourselves to graphs of maximum degree 3.

## 3. High-level description of the algorithm

A pseudo perfect matching is basically a spanning subgraph of $G$, which we will use as a starting point to constructing a perfect matching of $G$. Our algorithm for finding a perfect matching is based on finding a pseudo perfect matching and constantly "improving" it until a perfect matching is obtained.

The way we "improve" our pseudo perfect matching is using augmenting cycles. Each time we xor an augmenting cycle to the pseudo perfect matching we get closer to a perfect matching, since its number of edges decreases by at least one. Note that if the input graph has maximum degree 3, then the number of edges decreased when augmenting, equals the number of 3-vertices decreased. The following theorem shows that a good augmenting cycle exists whenever the pseudo perfect matching is not yet a perfect matching.

**Theorem 3.1** *Let $G$ be an undirected graph with maximum degree 3 having a perfect matching. Let $M$ be a pseudo perfect matching of $G$. If $M$ is not a perfect matching, then there exists a good augmenting cycle in $G$.*

**Proof:** Let $N$ be a perfect matching of $G$. Examine $M \oplus N$. This graph, excluding the isolated vertices, is a collection of vertex-disjoint cycles. Since $|M| > |N|$ there exists a cycle with more edges from $M$ than from $N$. This cycle is an augmenting one. It is moreover a good augmenting cycle, since $N$ is a matching. ■

We present below a high-level description of our $\mathcal{NC}$ algorithm for the perfect matching search problem on bipartite cubic graphs. Such graphs are known to have a perfect matching (See [12]).

Sections 4, 5, 6 and 7 describe the different stages of the algorithm. All stages, but the last one (section 7), can be applied to general graphs as well using the previously mentioned reduction to graphs with maximum degree 3.

**Notation 3.2** *Let $G$ be a graph with maximum degree 3. Let $M$ be a pseudo perfect matching of $G$. We denote by $n(M)$ the number of 3-vertices in $M$.*

**Perfect matching algorithm:**
**Input:** A bipartite 3-regular graph $G$.
**Output:** A perfect matching $M$ of $G$.

**begin**
    /* Section 4 */
    Construct a pseudo perfect matching of $G$
    and denote it by $M$.
    /* Section 5 */
    Convert $M$ to a forest in $G$.
    /* Section 6 */
    Convert $M$ to an induced forest in $G$.
    /* Section 7 */
    **while** $M$ is not a matching **do**:
    **begin**
        Find a perfect matching $N$
        in the complement of $M$.
        Find an augmenting cycle $L$ in $M \cup N$,
        such that $n(M \oplus L) \leq (1 - c)n(M)$,
        for a constant $c > 0$.
        $M = M \oplus L$.
    **end**
**end**

## 4. Constructing a pseudo perfect matching

This section describes the first stage of our algorithm, namely constructing a pseudo perfect matching of a given graph. Note that if the input graph is 3-regular (as in our case), we can take the whole graph as our initial pseudo perfect matching.

Let $G = (V, E)$ be a graph with $n$ vertices and $m$ edges. We define below a system of equations over $GF(2)$, whose solution is a pseudo perfect matching of $G$.

Let us assign for each edge $e \in E$ a variable $X_e \in \{0, 1\}$. The set of equations is as follows:

$$\forall v \in V : \oplus_{\{e:v \in e\}} X_e = 1$$

**Lemma 4.1** *Let $\vec{X}$ be a solution to this system of $n$ equations. Define $M(\vec{X}) = \{e \in E \mid \vec{X}_e = 1\}$, then $M(\vec{X})$ is a pseudo perfect matching of $G$.*

**Theorem 4.2** *Let $G$ be a graph with $n$ vertices, $m$ edges and maximum degree 3. If $G$ has a pseudo perfect matching, then there is an $\mathcal{NC}$ algorithm to find one, working in time $O(\log^2 n)$ using $O(n^{5.5})$ processors.*

**Proof:** **Correctness:** By our assumptions a solution to this linear system of equations exists, since a pseudo perfect matching solves it and $G$ has one. By the previous lemma any solution is a pseudo perfect matching, and correctness follows.

**Complexity:** The complexity of this algorithm is essentially the complexity of solving a linear system of $n$ equations with $m$ variables over $GF(2)$. This can be done using Mulmuley's algorithm for rank computation [14] in $O(\log^2 n)$ time using $O(n^{5.5})$ processors (see [8]). ■

## 5. Converting a pseudo perfect matching into a forest

This section and the next one deal with the problem of converting a pseudo perfect matching of a graph $G$ into one which is also an induced forest in $G$. The motivation for this transformation is to simplify the structure of the pseudo perfect matching as much as possible.

**Definition 5.1** *An* **Odd Forest** *of a graph $G$, is a pseudo perfect matching which is a forest in $G$.*

**Definition 5.2** *An* **Odd Induced Forest** *of a graph $G$, is a pseudo perfect matching which is an induced forest in $G$.*

Our algorithm to convert a pseudo perfect matching into an odd forest relies on the following lemma.

**Lemma 5.3** *Let $K$ be a graph. Let $S$ denote a set of fundamental cycles of $K$ with respect to some spanning forest of $K$. Define $K' = K \oplus (\oplus_{c \in S} c)$, then $K'$ is cycle free.*

**Proof:** Every non-forest edge lies in a unique fundamental cycle of $S$. When we xor the set of fundamental cycles to $K$ all non-forest edges vanish and only maybe some tree edges remain. It follows that $K'$ is cycle-free. ■

**Notation 5.4** *Let $G$ be a graph. Let $T$ be a spanning forest of $G$. For an edge $e \in E(G)$ denote by $P(e, T, G)$ the parity of the number of fundamental cycles with respect to $T$, which include $e$. For a vertex $v \in V(G)$ denote by $p(v)$ the parity of the number of non-forest edges incident on $v$.*

**Notation 5.5** *Let $T$ be a rooted tree. For an edge $e = (u, v) \in E(T)$, $v$ being the child of $u$, denote by $T(e)$ the set of vertices in the subtree rooted at $v$.*

**Algorithm for constructing an odd forest:**
**Input:** A graph $G$ with $n$ vertices and maximum degree 3;
A pseudo perfect matching $M$ of $G$.
**Output:** An odd forest $M$ of $G$.

**begin**
Find a spanning forest $T$ of $G$.
/* Xor to $M$ a set of fundamental cycles of $G$ */
In parallel for every edge $e \in E$ **do**:
**If** $P(e, T, G) = 1$ **then**
$M = M \oplus e$
**end**

**Lemma 5.6** *Let $G$ be a graph, and let $T$ be a spanning forest of $G$. If $e \in E(T)$ then*

$$P(e, T, G) = \oplus_{w \in T(e)} p(w)$$

**Proof:** Let $e = (u, v) \in E(T)$, $v$ being the child of $u$ in $T$. $P(e, T, G)$ is actually equal to the parity of the number of edges with one end vertex in $T(e)$ and the other in $V(T) - T(e)$, i.e. edges going out of the subtree rooted at $v$. The lemma now follows, since edges with both end vertices in $T(e)$ do not contribute to the right hand-side exclusive-or. ■

**Theorem 5.7** *There is an $\mathcal{NC}$ algorithm to convert a pseudo perfect matching into an odd forest in $O(\log n)$ time using $O(n)$ processors.*

**Proof:** **Correctness:** Applying the algorithm is equivalent to computing a set of fundamental cycles of $G$ with respect to $T$ and xoring them to $M$. By lemma 5.3, the graph obtained this way is cycle-free. Since xoring cycles

to a pseudo perfect matching preserves its structure, the result is an odd forest.

**Complexity:** An arbitrary spanning forest of $G$ can be computed in $O(\log n)$ time using $O(n)$ processors (see [6]).

It remains to show how to compute for a graph $G$, a spanning forest $T$ and an edge $e \in E(G)$, the value $P(e,T,G)$ efficiently. If $e$ is a non-forest edge then $P(e,T,G) = 1$. Otherwise, by the previous lemma if $e = (u,v)$, $v$ being the child of $u$, then $P(e,T,G)$ equals the xor over all vertices in the subtree rooted at $v$, of the parity of non-forest edges incident on those vertices. This computation can be made using a tree contraction algorithm in $O(\log n)$ time and $O(n/\log n)$ processors (see [6]).

The overall time of the algorithm is therefore $O(\log n)$ using $O(n)$ processors. ■

## 6. Converting an odd forest into an induced one

Let $M$ be an odd forest of a graph $G$. We describe below an algorithm to convert $M$ into an odd induced forest.

**Definition 6.1** *Let $T$ be a rooted tree in $M$ with $t > 4$ vertices. A $\frac{1}{3}-\frac{2}{3}$* **cut-vertex** *of $T$, is a vertex $v$, such that $v$ has at least $\frac{t-1}{3}$ descendants and at most $\frac{2t}{3}$ descendants, including itself. If $|V(T)| = 4$ define a cut-vertex of $T$ to be its 3-vertex.*

**Lemma 6.2** *Let $T$ be a tree in $M$ with $t > 2$ vertices. Root $T$ arbitrarily at a 3-vertex, then $T$ contains a $\frac{1}{3}-\frac{2}{3}$ cut-vertex.*

**Notation 6.3** *Let $G = (V,E)$ be a graph, and let $U \subset V$. Denote by $G[U]$ the graph induced by $G$ on the set of vertices $U$. For a vertex $v \in V$ denote by $S(v)$ the set of vertices which includes $v$ and all vertices adjacent to $v$.*

**Algorithm for constructing an odd induced forest:**
**Input:** A graph $G$ with $n$ vertices and maximum degree 3;
An odd forest $M$ in $G$.
**Output:** An odd induced forest $M$ of $G$.

**Induce($M$):**
**begin**
**while** $M$ is not induced,
in parallel for every tree $T$ in $M$
which is not induced **do**:
**begin**
Root $T$ arbitrarily at a 3-vertex.
Find a $\frac{1}{3}-\frac{2}{3}$ cut-vertex of $T$, denote it by $v$.
Search for a cycle $C$ in $G[T]$
which includes $v$.
**If** such a cycle $C$ exists **then**
$M = M \oplus C$
**else**
**begin**
$I$ =Induce$(T - \{v\})$
$M = (M - T) \cup I \cup T[S(v)]$
**end**
**end**
**end**

**Lemma 6.4** *The recursion depth of the algorithm is at most $\lceil \log_{3/2} n \rceil$.*

**Proof:** We will show that at each recursive call, the size of any tree component of $M$ which is not induced, decreases by at least a factor of $\frac{2}{3}$. Since a tree with two vertices must be induced, the lemma will follow.

Let $T$ be a tree which is not induced at the beginning of an iteration. Let $v$ denote a $\frac{1}{3}-\frac{2}{3}$ cut-vertex of $T$. The algorithm differentiates between two cases:

Case a) There exists a cycle $C$ in $G[T]$ which passes through $v$. In that case $T \oplus C$ comprises of two or more connected components and the size of each is at most $\frac{2}{3}|V(T)|$, since $v$ is a $\frac{1}{3}-\frac{2}{3}$ cut-vertex.

Case b) There is no such cycle. In this case, removing $v$ we get three connected components, each of size at most $\frac{2}{3}|V(T)|$.

At the $k$'th iteration therefore, all trees which are not induced are of size at most $(\frac{2}{3})^k n$. The lemma follows. ■

**Theorem 6.5** *There is an $\mathcal{NC}$ algorithm to convert an odd forest into an induced one in $O(\log^2 n)$ time using $O(n\alpha(n)/\log n)$ processors.*

**Proof:** **Correctness:** By the previous lemma the algorithm outputs an induced forest. Since xoring cycles to $M$ preserves its structure as a pseudo perfect matching, we obtain an odd induced forest.

**Complexity:** The algorithm runs at most $\lceil \log_{3/2} n \rceil$ iterations (iteration being a level of recursive call). Each iteration includes finding the connected components of $M$ and some tree computations done for every tree in $M$.

The connected components of $M$ can be found in $O(\log n)$ time using $O(n\alpha(n)/\log n)$ processors, where $\alpha(n)$ is the inverse Ackermann function (see [2]).

Given a tree $T$ with $t$ vertices, we can use the Euler tour technique (see [17]) to perform our computations on $T$. Rooting $T$ can be done in $O(\log t)$ time using $O(t/\log t)$ processors (see [6]). Finding a $\frac{1}{3}-\frac{2}{3}$ cut vertex of $T$ can be done by computing for each vertex in $T$ its number of descendants and then determining a cut vertex. This computation can be performed in $O(\log t)$ time using $O(t/\log t)$ processors (see [6]). Checking if a given vertex is on a cycle and reconstructing such a cycle requires $O(\log t)$ time using $O(t/\log t)$ processors.

The overall time of the algorithm is therefore $O(\log^2 n)$ using $O(n\alpha(n)/\log n)$ processors. ■

## 7. Computing a perfect matching

This section describes the last stage of our algorithm. We show how to compute a perfect matching given an odd induced forest. The algorithm consists of $O(\log n)$ iterations. The input to each iteration is an odd induced forest of $G$ and the output is a new odd induced forest in which the number of 3-vertices is only a constant fraction of the original one. This new forest is obtained by augmenting with a good cycle, passing through a constant fraction of the 3-vertices in the input forest.

**Definition 7.1** *Let $G$ be a graph. An open* **2-Path** *in $G$, is an open path in which all internal vertices have degree 2 and its end vertices are of degree other than 2.*

**Algorithm for constructing a perfect matching:**
**Input:** A bipartite 3-regular graph $G$ with $n$ vertices;
An odd induced forest $M$ of $G$.
**Output:** A perfect matching $M$ of $G$.

**begin**
**While** $M$ is not a perfect matching **do**:
**begin**
Compute a perfect matching $N$
in the complement of $M$.
Let $H = M \cup N$.
Compute a spanning forest $T$ of $H$.
/* Xor to $M$ a set of fundamental
cycles of $H$ */
In parallel for every edge $e \in E(T)$ **do**
**If** $P(e, T, H) = 1$ **then**
$M = M \oplus e$.
**end**
**end**

The first step in each iteration of the algorithm is computing a perfect matching in the complement of $M$. A perfect matching of the complement exists and is easily found by the following lemma.

**Lemma 7.2** *Let $G$ be a bipartite 3-regular graph. Let $M$ be a pseudo perfect matching of $G$, then the complement of $M$ in $G$ is a collection of vertex-disjoint even cycles.*

Denote this perfect matching by $N$. Let $H$ denote the graph $M \cup N$.

**Lemma 7.3** *Let $Q$ be a 2-path in $H$. If $e, f \in Q$ then $P(e, T, H) = P(f, T, H)$.*

**Corollary 7.4** *Let $Q$ be a 2-path in $H$ at a beginning of an iteration, its two end-vertices being $u$ and $v$. If $e \in Q$ and $P(e, T, H) = 1$ then $u$ and $v$ will be removed as 3-vertices from $M$ at the end of that iteration.*

**Proof:** We will prove for $u$, the same is valid for $v$. Let $f \in Q$ be an edge incident on $u$ (maybe $e = f$). By the previous lemma $P(f, T, H) = 1$ since $P(e, T, H) = 1$. It follows that $f$ is on an augmenting cycle in $H$ and this cycle includes $u$. At the end of the iteration therefore $u$ will be removed as a 3-vertex from $M$. ■

Let $S$ denote a set of fundamental cycles in $H$ with respect to $T$. Let $L$ denote the exclusive-or of all these cycles.

**Proposition 7.5** *Let $M' = M \oplus L$, then $n(M') \leq \frac{1}{2} n(M)$.*

**Proof:** Let $I$ be any connected component of $H$ having $2p$ 3-vertices and $2q$ vertices of degree 2. The dimension of its cycle space is $3p + 2q - (2p + 2q) + 1 = p + 1$. It follows that $p + 1$ non-forest edges of $I$ lie in $L$.

Let $e$ be such a non-forest edge. If $e$ is incident on two 3-vertices, then both will be removed when augmenting with a cycle including $e$. Otherwise, $e$ must lie in a 2-path $Q$ of $H$. By corollary 7.4 both end-vertices of $Q$ will be removed as 3-vertices when augmenting with a cycle including $Q$. We conclude that $L$ contains at least $p+1$ 3-vertices of $I$.

The proposition now follows, noting that the last statement is true for every connected component of $H$, and that every augmenting cycle in $H$ is a good one (this is simply because no two adjacent edges from the complement of $M$ reside in $H$). ■

**Theorem 7.6** *Let $G$ be a bipartite cubic graph. Given an odd induced forest $M$ in $G$, there is an $\mathcal{NC}$ algorithm to obtain a perfect matching of $G$, working in time $O(\log^2 n)$ using $O(n\alpha(n)/\log n)$ processors.*

**Proof:** **Correctness:** Each iteration we augment $M$, using a good augmenting cycle, at least halving the number of 3-vertices in $M$ by proposition 7.5. The algorithm therefore terminates after at most $\lceil \log n \rceil$ iterations with $M$ being a perfect matching of $G$.

**Complexity:** By proposition 7.5, $\lceil \log n \rceil$ iterations suffice to produce a perfect matching. Each iteration consists of the following: Computing a perfect matching in the complement; computing a spanning forest $T$ of $H$; and computing $P(e, T, H)$, for every every edge $e \in E(T)$.

Computing a perfect matching in the complement can be done in time $O(\log n)$ using $O(n\alpha(n)/\log n)$ processors, by finding its connected components (see [2]) and taking alternating edges from each of them.

If $M$ has $k$ 3-vertices then Computing a spanning forest of $H$ and the values of $P(e, T, I)$ for every $e \in E(T)$ can be done, as shown in theorem 5.7, in $O(\log n)$ time and $O(k + n/\log n)$ processors. Since the number of 3-vertices in $M$ decreases by half each iteration, the total work required for these computations is $O(n \log n)$.

The overall time of the algorithm is therefore $O(\log^2 n)$ using $O(n\alpha(n)/\log n)$ processors. ■

**Theorem 7.7** *There is an $\mathcal{NC}$ algorithm for the perfect matching problem restricted to bipartite cubic graphs, working in $O(\log^2 n)$ time and $O(n\alpha(n)/\log n)$ processors.*

**Proof:** **Correctness:** We start with the input graph (which is 3-regular) as our initial pseudo perfect matching. Correctness now follows from theorems 5.7, 6.5 and 7.6.

**Complexity:** By theorems 5.7, 6.5 and 7.6, the overall time of the algorithm is $O(\log^2 n)$ using $O(n\alpha(n)/\log n)$ processors. ■

**Theorem 7.8** *A sequential implementation of our algorithm runs in $O(n \log n)$ time.*

**Proof:** The bottleneck in the complexity of the algorithm is computing connected components over $O(\log n)$ iterations in the algorithms of sections 6 and 7. Since serially the connected components of a graph with $n$ vertices can be computed in $O(n)$ time we obtain the stated bound. ■

## 8. Concluding remarks

We presented a deterministic algorithm for the perfect matching problem on bipartite cubic graphs working in $O(\log^2 n)$ time using $O(n\alpha(n)/\log n)$ processors. Some further results for general graphs are presented in [16]. The question whether the perfect matching problem is in $\mathcal{NC}$ remains open.

## References


[1] N. Blum. A new approach to maximum matching in general graphs. In M. Paterson, editor, *ICALP 90: Automata, Languages, and Programming (LNCS 443)*, pages 586–597, 1990.

[2] R. Cole and U. Vishkin. Approximate parallel scheduling. Part II: Applications to logarithmic-time optimal graph algorithms. *Information and Computation*, 92:1–47, 1991.

[3] E. Dahlhaus and M. Karpinski. Perfect matching for regular graphs is $\mathcal{AC}^0$-hard for the general matching problem. *J. Comput. and System Sci.*, 44(1):94–102, 1992.

[4] J. Edmonds. Paths, trees and flowers. *Canada J. Math*, 17:449–467, 1965.

[5] D. Grigoriev and M. Karpinski. The matching problem for bipartite graphs with polynomially bounded permanents is in $\mathcal{NC}$. In *The 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 166–172, 1987.

[6] J. Ja'Ja'. *An introduction to parallel algorithms*. Addison-Wesely Publishing Company, 1992.

[7] H. J. Karloff. A las vegas $\mathcal{NC}$ algorithm for maximum matching. *Combinatorica*, 6(4):387–391, 1986.

[8] R. M. Karp and R. Ramachandran. A survey of parallel algorithms for shared-memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 17. MIT Press, Cambridge, Mass., 1990.

[9] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a maximum matching is in random $\mathcal{NC}$. *Combinatorica*, 6:35–48, 1986.

[10] A. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. on Computers*, 30(2):93–100, 1981.

[11] L. Lovasz. On determinants, matchings and random algorithms. In L. Budach, editor, *Fundamentals of Computing Theory*. Akademia-Verlag, Berlin, 1979.

[12] L. Lovasz and M. D. Plummer. *Matching Theory*, pages 122–126. Academic Press, Budapest, Hungary, 1986.

[13] S. Micali and V. V. Vazirani. An $O(\sqrt{|V||E|})$ algorithm for finding maximum matching in general graphs. In *The 21th Annual IEEE Symposium on Foundations of Computer Science*, pages 17–27, 1980.

[14] K. Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, 7:101–104, 1987.

[15] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.

[16] R. Sharan. Perfect matching in parallel computation. Master's thesis, Institute of Computer Science, Hebrew University of Jerusalem, Israel, 1995.

[17] R. E. Tarjan and U. Vishkin. An efficient biconnectivity algorithm. *Siam J. Computing*, 14(4):862–874, 1985.

[18] W. T. Tutte. The factorization of linear graphs. *J. London Math Society*, 22:107–111, 1947.

[19] V. V. Vazirani. $\mathcal{NC}$ algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems. *J. Information and Computation*, 80, 1989.

[20] V. V. Vazirani. A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{|V||E|})$ general graph maximum matching algorithm. *Combinatorica*, 14, 1994.