Some perfect matchings and perfect half-integral matchings in NC*

Raghav Kulkarni†
Department of Computer Science
University of Chicago
Chicago, USA
Email: raghav@cs.uchicago.edu

Meena Mahajan‡
The Institute of Mathematical Sciences
Chennai 600 113, India
Email: meena@imsc.res.in

Kasturi R. Varadarajan§
Department of Computer Science
The University of Iowa
Iowa City, IA 52242-1419 USA
Email: kvaradar@cs.uiowa.edu

September 4, 2008

Abstract

We show that for any class of bipartite graphs which is closed under edge deletion and where the number of perfect matchings can be counted in NC, there is a deterministic NC algorithm for finding a perfect matching. In particular, a perfect matching can be found in NC for planar bipartite graphs and $K_{3,3}$-free bipartite graphs via this approach. A crucial ingredient is part of an interior-point algorithm due to Goldberg, Plotkin, Shmoys and Tardos. An easy observation allows this approach to handle regular bipartite graphs as well.

We show, by a careful analysis of the polynomial time algorithm due to Galluccio and Loebl, that the number of perfect matchings in a graph of small ($O(\log n)$) genus can be counted in NC. So perfect matchings in small genus bipartite graphs can also be found via this approach.

We then present a different algorithm for finding a perfect matching in a planar bipartite graph. This algorithm is substantially different from the algorithm described above, and also from the algorithm of Miller and Naor, which predates the approach of Goldberg *et al.* and tackles the same problem. Our new algorithm extends to small genus bipartite graphs, but not to $K_{3,3}$-free bipartite graphs. We next show that a non-trivial extension of this algorithm allows us to compute a vertex of the fractional perfect matching polytope (such a vertex is either a perfect matching or a half-integral matching) in NC, provided the graph is planar or small genus but *not necessarily bipartite*, and has a perfect matching to begin with. This extension rekindles the hope for an NC-algorithm to find a perfect matching in a non-bipartite planar graph.

*Most results in this paper were originally announced in papers in Proc. 32nd ACM Symposium on Theory of Computing (2000) [26] and Proc. 12th European Symposium on Algorithms (2004) [19].

†Parts of this work were done when this author was at the Chennai Mathematical Institute, and while visiting the Institute of Mathematical Sciences, Chennai.

‡Part of this work was done when this author was supported by the NSF grant CCR-9734918 on a visit to Rutgers University, USA.

§Parts of this work were done when this author was at DIMACS, Rutgers University, USA and then at the Tata Institute of Fundamental Research, Mumbai, India.

1 Introduction

The perfect matching problem is of fundamental interest in combinatorics, algorithms and complexity theory for a variety of reasons. In particular, the problems of deciding if a graph has a perfect matching, and finding such a matching if one exists, have received considerable attention in the field of parallel algorithms. (See for instance [36, 15].) Both these problems are in randomized NC [22, 14, 29] but are not known to be in deterministic NC. For special classes of graphs, however, there are deterministic NC algorithms.

The outstanding open problem here is to give a deterministic NC algorithm for detecting/searching for a perfect matching in a given graph. Some progress in this direction was achieved with two sublinear time parallel algorithms for bipartite matching: $O(n^{2/3}\log^3 n)$ due to Goldberg, Plotkin and Vaidya [10], and $O(m^{1/2}\log^3 n)$ due to Goldberg, Plotkin, Shmoys and Tardos [9]. NC algorithms have been obtained for some special classes of graphs; for instance, dense graphs (minimum degree at least $n/2$) [3], regular bipartite graphs [20], strongly chordal graphs [4], graphs with polynomially bounded number of perfect matchings [11], P_4-tidy graphs[30], convex bipartite graphs [6], incomparability graphs [17], claw-free graphs [2], and planar bipartite graphs [27].

The problem of counting perfect matchings in a given graph is #P-hard [34], and a polynomial time algorithm is therefore unlikely for it. Again, for special classes of graphs, counting is easy; for instance, planar graphs, $K_{3,3}$-free graphs (graphs that do not contain a subgraph homeomorphic to $K_{3,3}$; this class of graphs includes all planar graphs), regular bipartite graphs. The category we are specifically interested in is planar graphs. A combination of results due to Kasteleyn [16], Little [21] and Vazirani [35] establishes that for planar graphs, counting perfect matchings can be done in P and even in NC. The technique is to establish that every planar graph has what is called a pfaffian orientation, one in which every perfect matching gets the same sign under a suitable signing convention. The signed sum of all perfect matchings is easy to compute; thus given a pfaffian orientation the number of perfect matchings can be obtained. Recently, Galluccio and Loebl [8] extended the results of Kasteleyn to the case of graphs of small genus. They proved that the generating function of the perfect matchings of a graph of genus g may be obtained as a linear combination of 4^g pfaffians. This means that one can count perfect matchings by first finding 4^g orientations, and then taking a linear combination of 4^g corresponding determinants. This gives a polynomial time algorithm for any n-vertex graph graph with $O(\log n)$ genus.

Surprisingly, an NC algorithm for finding a perfect matching in a planar graph has proved quite elusive. This situation is rather intriguing, as it contradicts our intuition that search should be easier than counting. For the case of bipartite planar graphs, Miller and Naor [27] succeeded in giving an NC algorithm to find a perfect matching. Their method reduces this problem to that of finding a circulation in the planar graph, which then corresponds to computing shortest paths in its planar dual.

We expect the search problem to be easier for classes of bipartite graphs rather than general graphs because of a fundamental property of these graphs: the fractional perfect matching polytope $\mathcal{FPM}(G)$ (see Section 2 for definitions) for bipartite graphs coincides with the perfect matching polytope $\mathcal{PM}(G)$. This breaks down in general graphs; $\mathcal{PM}(G)$ could even be empty while $\mathcal{FPM}(G)$ is non-trivial. Thus, for instance, we see no way of extending the algorithm of [27], or even the algorithm of [9], to planar non-bipartite graphs.

Our contributions are as follows:

1. We show in Section 3 that for bipartite graphs, search is not harder than counting. For any

class of bipartite graphs where counting perfect matchings is in NC (and the class is closed under edge deletion), finding a perfect matching is also in NC. This result is established using the second part of the algorithm of [9], and it applies to bipartite graphs that are $K_{3,3}$-free or regular. To the best of our knowledge, this connection between counting and the applicability of [9] has not been made before.

2. We show that the number of perfect matchings in a graph of genus $O(\log |V|)$ can be computed in NC. This is established by extending the algorithm of Galluccio and Loebl [8], and using the algorithm of [25] to compute the pfaffian with its sign in NC. This result is established in Section 4.

3. We give, in Section 5, a new NC algorithm for finding a perfect matching in a planar bipartite graph. This algorithm is essentially different from that of Section 3, as well as from that of [27]. It extends to small genus bipartite graphs as well, though not to $K_{3,3}$-free bipartite graphs.

4. Using the algorithm of Section 5 as a starting point, we devise in Section 6 an NC algorithm to find a vertex of the fractional perfect matching polytope $\mathcal{FPM}(G)$ of a planar, *not necessarily bipartite*, graph. As in the the algorithm of Section 5, this algorithm is also an interior-point algorithm; it stays within $\mathcal{FPM}(G)$ while navigating towards a vertex. Using the result of Section 4, it extends to small-genus graphs as well. This is one of the first results of this flavor - finding the optimum of a linear program instance in NC when the corresponding integer program is solvable in P. This algorithm also rekindles the hope of obtaining an NC algorithm for finding a perfect matching in a planar graph.

2 Preliminaries

2.1 Matching polytopes

Let G be a graph with m edges. Consider the polytope $\mathcal{FPM}(G)$ in m-dimensional space, defined by the following constraints.

$$\begin{aligned} x_e &\geq 0 \quad \forall\, e \in E \\ \sum_{e \text{ incident on } v} x_e &= 1 \quad \forall\, v \in V \end{aligned} \tag{1}$$

This polytope is called the fractional perfect matching polytope. Clearly, every perfect matching of G (i.e. the corresponding point in $\mathcal{Q}^m$) lies in $\mathcal{FPM}(G)$. Standard matching theory (see for instance [23]) tells us that every perfect matching of G is a vertex of $\mathcal{FPM}(G)$. (A perfect matching is an integral solution to Equations 1). In the case of bipartite graphs, the perfect matchings are in fact the only vertices of $\mathcal{FPM}(G)$; $\mathcal{FPM}(G) = \mathcal{PM}(G)$. Thus for a bipartite graph it suffices to find a vertex of $\mathcal{FPM}(G)$ to get a perfect matching. Furthermore, for general graphs, all vertices of $\mathcal{FPM}(G)$ are always half-integral (in the set $\{0, 1/2, 1\}^m$). For any vertex w of $\mathcal{FPM}(G)$, if we pick those edges of G having non-zero weight in w, we get a subgraph which is a disjoint union of a partial matching and some odd cycles.

For instance, Figure 1 shows a graph where $\mathcal{PM}(G)$ is empty, while $\mathcal{FPM}(G)$ has one point shown by the weighted graph alongside. Figure 2 shows a graph where $\mathcal{PM}(G)$ is non-empty; furthermore, the weighted graph in Figure 2(b) is a vertex of $\mathcal{FPM}(G)$ not in $\mathcal{PM}(G)$.

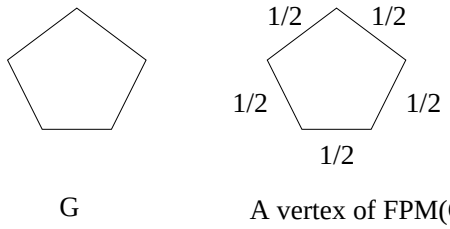

Figure 1: An Example Graph with empty $\mathcal{PM}(G)$, and a point in $\mathcal{FPM}(G)$

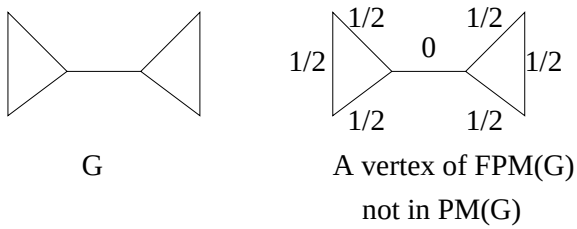

Figure 2: An Example Graph with non-empty $\mathcal{PM}(G)$, and a point in $\mathcal{FPM}(G) - \mathcal{PM}(G)$

The matching polytope $\mathcal{M}(G)$ for a graph G is the convex hull of all matchings (not necessarily perfect) in G. All maximal matchings are vertices of $\mathcal{M}(G)$. If G is bipartite, then the polytope $\mathcal{M}(G)$ is also defined by the following equations.

$$\begin{aligned} x_e &\geq 0 \quad \forall\, e \in E \\ \sum_{e \text{ incident on } v} x_e &\leq 1 \quad \forall\, v \in V \end{aligned} \tag{2}$$

That is, for bipartite graphs, all vertices of the polytope defined by Equations 2 are integral and hence correspond to matchings in G. Thus for bipartite graphs, $\mathcal{PM}(G) = \mathcal{FPM}(G) \subseteq \mathcal{M}(G)$.

2.2 Embeddings

A planar embedded graph G partitions the plane into connected regions called *faces*. Let F denote the set of faces of G. Let $G^* = (F, E^*)$ be the *dual graph* of G. Its vertex set is the set F of faces of G. There is a one-to-one correspondence between its edge set E^* and E as follows: for each edge $e \in E$, there is a corresponding edge $e^* \in E^*$ connecting the two faces in G that border e. Note that G^* may have loops and multi-edges even when G does not. However, G^* is always a connected graph, irrespective of whether or not G is. We sometimes refer to G as the *primal graph*. We further denote by G^{**} the graph obtained from G^* by deleting self-loops and replacing multiple edges by a single edge. Let n, m and f (respectively n^*, m^*, f^*, and n^{**}, m^{**}, f^{**}) denote the number of vertices, edges and faces in G (respectively G^* and G^{**}), and let c be the number of components in G. Then $e^{**} \leq e^* = e$, $f^{**} \leq f^* = n - c + 1$, and $n^{**} = n^* = f$. Also, $f \in O(e)$.

Euler's formula states that a planar graph with c components satisfies $e + c + 1 = n + f$. Furthermore, if the graph has no self-loops or multiple edges, it also satisfies $e \leq 3n - 6c$.

These definitions generalize naturally if the graph G is embedded on an orientable surface of higher genus. Throughout this paper, surface will mean orientable surface. The genus of a graph G, denoted $\gamma(G)$, is the minimum g for which G can be embedded on a surface S_g of genus g. If the graph is connected and has no self-loops or multiple edges, it satisfies $e \leq 3n + 6(\gamma(G) - 1)$.

Given an embedding of a graph G on a surface S_g, an edge is said to be a *separating edge* if the two faces of the embedding incident on e are distinct from each other. A cycle C in G is said to be a *separating cycle* if some edge e of C is a separating edge. In a planar embedding, every cycle is separating, but this may not be the case for higher genus embeddings.

A combinatorial embedding is a scheme that specifies, for each vertex v, a cyclic permutation of the neighbors of v. For connected graphs, combinatorial embeddings are in one-to-one correspondence with embeddings on surfaces of appropriately calculated genus.

For a full treatment of this topic, see for instance [33, 38].

2.3 The computation model

Most algorithms in this paper are presented using the concurrent-read concurrent-write (CRCW) parallel random-access machine PRAM model, see for instance [13]. To show that a problem is in NC, we describe a parallel algorithm that, for inputs of size n, uses $n^{O(1)}$ processors and runs in $\log^{O(1)} n$ parallel time. These processors access a global shared memory consisting of $n^{O(1)}$ memory locations.

For the reduction in Section 3, we need the notion of oracle access within NC. For this result alone, we use the equivalent formulation of NC as problems decided by circuit families $\{C_n\}_{n>0}$, where C_n is a Boolean circuit with $n^{O(1)}$ AND, OR and NOT gates, depth $\log^{O(1)} n$, and decides instances of length n. Further, given 1^n as input, a description of C_n can be obtained using $\log n$ space. A circuit family relative to an oracle B (a decision problem) is also allowed to have query gates, with each such gate having m inputs for some $m \in n^{O(1)}$. A query gate outputs a 1 if the word formed by the settings to its input wires is in the oracle language B, and outputs 0 otherwise. See, for instance, [37] for more details. A problem A is said to NC-reduce to a decision problem B, denoted $A \in \mathrm{NC}(B)$, if A is decided by an NC circuit family relative to oracle B.

3 Bipartite graphs: Search reduces to counting

The main result of this section is that for any subclass of bipartite graphs closed under deletion, searching for a perfect matching NC-reduces to counting perfect matchings. Formally, Let $\mathcal{F}$ be any class of bipartite graphs closed under edge deletion. Define the problems

SEARCH-PM$_{\mathcal{F}}$: On input $G \in \mathcal{F}$, output a perfect matching M in G, if one exists.

COUNT-PM$_{\mathcal{F}}$: On input $G \in \mathcal{F}$, output the number of perfect matchings in G.

BIT-COUNT-PM$_{\mathcal{F}}$. This is the decision problem

$$\{\langle G, i, b\rangle \text{ the } i\text{th bit of COUNT-PM}_{\mathcal{F}}(G) \text{ is } b\ \}$$

Theorem 1 *For any class $\mathcal{F}$ of bipartite graphs closed under edge deletion,*

$$\text{SEARCH-PM}_{\mathcal{F}} \in \mathrm{NC}(\text{BIT-COUNT-PM}_{\mathcal{F}})$$

A major tool in proving this result is the technique developed in [9], where Goldberg *et al.* describe a parallel algorithm to construct a maximum weight matching in a weighted bipartite graph. The algorithm has two major stages. In the first stage, a feasible solution (an interior point of $\mathcal{M}(G)$) and a corresponding dual solution with a small duality gap are obtained. This interior point is in fact near-optimal; the weight of the feasible solution is less than the weight of an optimal solution by at most 1/2. In the second stage, the fractional values of the near-optimal point are carefully rounded to obtain an optimal integral solution. The first stage needs $O(\sqrt{m}\log^2 n \log nC)$

parallel time while the second stage needs $O(\log n \log nC)$ parallel time, both using polynomially many processors. Here, C is the maximum absolute value of the weights of edges in G.

We are concerned here with the specialization of the above algorithm to unweighted bipartite graphs (all edges have weight 1), and with the problem of perfect matchings. Suppose the given graph G, on $2n$ vertices, has a perfect matching. Then we have

Proposition 2 (Section 3 of [9]) *Let G be a bipartite graph on $2n$ vertices with at least one perfect matching. An interior point of $\mathcal{M}(G)$ with total weight at least $n - 1/2$ can be found in parallel time $O(\sqrt{m}\log^3 n)$ using polynomially many processors.*

Note that this point is not in the interior of $\mathcal{PM}(G) = \mathcal{FPM}(G)$ unless it is already optimal in the weighted sense.

Proposition 3 (Section 4 of [9]) *Let G be a bipartite graph on $2n$ vertices with at least one perfect matching. From any interior point of $\mathcal{M}(G)$ with total weight at least $n - 1/2$, a perfect matching of G can be obtained in NC (parallel time $O(\log^2 n)$).*

Since Proposition 3 already gives an NC bound for navigating from a near-optimal interior point to a vertex of $\mathcal{M}(G)$, we can hope to obtain an NC algorithm for finding a perfect matching if the task of Proposition 2 can also be performed in NC. We observe below that for a large class of graphs, this is indeed the case. The crucial observation is the following:

Fact 4 *Every interior point of $\mathcal{FPM}(G)$ has total weight n and is thus optimal in the sense of [9].*

Lemma 1 *Let $\mathcal{F}$ be any class of graphs closed under edge deletion. For any graph $G \in \mathcal{F}$, an interior point of $\mathcal{PM}(G) \subseteq \mathcal{FPM}(G)$, if one exists, can be obtained in NC with oracle access to* BIT-COUNT-PM*$(\mathcal{F})$.*

Proof: For each edge $e \in G$, set x_e to be the ratio of the number of perfect matchings in which e participates to the total number of perfect matchings. To compute the values x_e, we use the oracle BIT-COUNT-PM$(\mathcal{F})$ as follows.

Compute M = the number of perfect matchings in G.
If $M = 0$, then report "$\mathcal{PM}(G) = \emptyset$" and exit.
For each $e \in E$, pardo:
    Compute M_e = the number of perfect matchings in $G - \{e\}$.
    $x_e = 1 - (M_e/M)$.
    Endif

It is clear that this point satisfies Equations 1 and hence is inside $\mathcal{FPM}(G)$. It is inside $\mathcal{PM}(G)$ as well, since it is simply the centroid of all perfect matchings and thus is inside their convex hull. ■

Note that the above result does not even require G to be bipartite.

Using Fact 4 and Lemma 1, we see that Proposition 2 can be partially tightened as follows:

Proposition 5 *Let $\mathcal{F}$ be any class of graphs closed under edge deletion. For any G in $\mathcal{F}$, if $\mathcal{PM}(G)$ is non-empty, then an interior point of $\mathcal{PM}(G)$ with total weight n can be found in NC with oracle access to* BIT-COUNT-PM*$(\mathcal{F})$.*

Since $\mathcal{FPM}(G) = \mathcal{PM}(G) \subseteq \mathcal{M}(G)$ for bipartite graphs, putting Proposition 5 and Proposition 3 together establishes Theorem 1.

For $K_{3,3}$-free graphs, it turns out that BIT-COUNT-PM is in NC:

Lemma 2 ([35]) *The number of perfect matchings in $K_{3,3}$-free graphs can be computed in NC.*

Proof: Given a $K_{3,3}$-free graph G, [35] shows how to obtain a pfaffian orientation of G, and [25] shows how to compute the pfaffian. The absolute value of this pfaffian is the number of perfect matchings in G. (Note: the absolute value of the pfaffian can also be computed by computing the square root of the determinant. [25] directly computes the pfaffian, with its sign, in NC.) ■

Hence, using Theorem 1, we can conclude:

Corollary 1 *There is an NC algorithm that, given a $K_{3,3}$-free bipartite graph G, finds a perfect matching in G (or reports that none exists).*

For regular bipartite graphs, interior points are even easier to compute. Let d be the degree of every vertex; simply set $x_e = 1/d$ for every edge e. Clearly, this assignment satisfies Equations 1.

Proposition 6 *If G is a regular bipartite graph, then an interior point of $\mathcal{FPM}(G) \subseteq \mathcal{M}(G)$ can be obtained in NC.*

Using Fact 4, Proposition 6, and Proposition 3, we have the following corollary.

Corollary 2 *There is an NC algorithm that, given a regular bipartite graph G, finds a perfect matching in G (or reports that none exists).*

4 Counting perfect matchings in small genus graphs in NC

In this section we show that the number of perfect matchings in a graph of genus $g \in O(\log n)$ can be computed in NC.

4.1 Input and basic operations

We first describe the *polygonal representation* of an oriented surface S_g of genus g. We follow the notation of [8]. Informally, S_g is a sphere with g handles, and its polygonal representation is obtained by cutting the g handles of its space model. The polygonal representation consists of a convex $4g$-gon B_0 whose vertices $a_1, \ldots, a_{4g}$ are numbered clockwise. For $0 \leq i \leq g-1$, the edge $[a_{4i+1}, a_{4i+2}]$ is identified with the edge $[a_{4i+4}, a_{4i+3}]$ via a bridge B_1^i, and the edge $[a_{4i+2}, a_{4i+3}]$ is identified with the edge $[a_{4i+5}, a_{4i+4}]$ via a bridge B_2^i. (The vertex a_{4g+1} is, by convention, the vertex a_1.) The bridges B_1^i and B_2^i together constitute the ith handle of the sphere. We will use $[a, b]$ to denote an edge of the convex polygon B_0 as opposed to an edge of the graph. We will refer to two edges of B_0 that are identified as *partners*.

Thus a graph G embedded on S_g, when translated to the polygonal representation, has its vertices inside B_0. An edge of G might "cross over" from an edge of B_0 to its partner using a

bridge. Our algorithm assumes some combinatorial representation of the embedding of G within B_0 (any scheme for representing planar graphs suffices), plus some additional information that identifies points on an edge of G when it crosses over from an edge of B_0 to its partner. Given such an embedding, we can easily modify existing NC-algorithms for planar graphs to perform standard operations like extracting the edges bounding each face of G, or constructing the graph G^* dual to G.

For more details concerning embeddings onto non-planar surfaces, see for instance [33, 38, 8].

4.2 Counting perfect matchings in P: the Galluccio-Loebl algorithm

We briefly describe the polynomial time algorithm of Galluccio and Loebl [8] for counting perfect matchings in G.

Galluccio and Loebl actually consider the following generalization of the counting problem. Let x_e be a distinct variable associated with each edge e of the graph G. The monomial corresponding to a perfect matching M is $x(M) = \prod_{e \in M} x_e$, the product of the variables corresponding to edges in M. Define $P(G, x)$ to be the sum of $x(M)$ over of all perfect matchings in the graph; this is called the *generating function of perfect matchings* of G. When all the variables are set to one, $P(G, x)$ evaluates to the number of perfect matchings in G.

Theorem 7 (Theorem 3.10 of [8]) *Let G be a graph embeddable on an orientable surface of genus g. Then $P(G, x)$ may be expressed as a linear combination of 4^g Pfaffians of matrices $A(D)$ where each D is an orientation of G.*

To prove this theorem, the notion of a proper g-graph is defined.

Definition 8 (Definition 2.2 of [8]) *A graph G is called a proper g-graph if it may be embedded on S_g satisfying the following conditions:*

1. *All vertices are embedded in B_0.*
2. *Each edge uses at most one of the $2g$ bridges.*
3. *Vertices on the boundary of B_0 form a cycle embedded on the boundary.*
4. *Endpoints of an edge that uses a bridge are embedded on the boundary of B_0, one vertex on each partner.*
5. *Each vertex has at most one incident edge not in G_0.*
6. *G_0 has a perfect matching M_0.*

Also, the proof uses the following convention. In some situations we want x_e to be hardwired to some number a_e for certain edges e. In this case, we instantiate x_e to a_e in the original polynomial, and we call the resulting polynomial the generating polynomial for the hardwired graph.

The proof of Theorem 7 involves three stages:

1. Obtain from G another graph G' such that $P(G, x) = P(G', x')$ and G' is a proper g-graph:
   (a) Start with any embedding of G on S_g. Without loss of generality, all vertices can be assumed to be in the interior of B_0.

(b) If an edge e uses multiple bridges, subdivide it at each point where it crosses the boundary of B_0. This creates an odd-length path $e_1, \ldots, e_{2k+1}$. This ensures that an edge uses a bridge at most once (with endpoints on partner edges), and that a vertex on the boundary of B_0 has at most one incident edge that uses a bridge. Let $x'_{e_1} = x_e$, and $x'_{e_j} = 1$ for $j > 1$.

(c) Fix a perfect matching M in G. (If none exists, then $P(G, x)$ is the zero polynomial.) After the above subdivisions, it yields a perfect matching M_0 within B_0.

(d) Add edges to form a cycle on the boundary of B_0. Let $x'_e = 0$ for such an edge e.

2. Express $P(G', x')$ as a linear combination of 4^g terms, each being the Pfaffian of a matrix $A(D')$, where D' is an orientation of G'.
This is established in Theorem 3.8 of [8]. The relevant orientations D' of G' are obtained as follows: find any basic Pfaffian orientation D'_0 of the (planar) subgraph G_0 that is G' restricted to B_0. Also, find Pfaffian orientations D'_i of each (planar) subgraph G_i that is G' restricted to B_0 and the ith bridge, for $i = 1, \ldots, 2g$. Let $-D'_i$ be the orientation obtained by reversing the orientation D'_i. The relevant orientations of G' are those that equal D'_0 in B_0, and for each $i = i, \ldots 2g$, equal either D'_i or $-D'_i$ on the ith bridge (Definitions 2.4, 2.5, 2.6 of [8]). Clearly there are 2^{2g} such orientations D', each with a signature $r(D') \in \{-1, +1\}^{2g}$. The coefficient for the Pfaffian of D' in the linear combination is computed from the signature $r(D')$ by a simple expression; see Definition 3.4 of [8].

3. Obtain for each relevant orientation D' of G' an orientation D of G with the same Pfaffian. To recover an orientation D of G, throw away edges with $x'_e = 0$. For an edge e of G that was subdivided into an odd path in G', orient it in the direction in which an odd number of edges of the subdivision are oriented in D'.

4.3 Counting perfect matchings

By modifying the algorithm of Galluccio and Loebl [8] described above, we obtain a parallel algorithm to count perfect matchings given a polygonal embedding of the graph G. Stage 2 and Stage 3 above are easily parallelizable. In particular, once the embedding is available, we can in parallel run 4^g algorithms, one for each possible signature $r(D)$. Each such algorithm computes the relevant orientation with a given signature, and its Pfaffian. By using the results from [35] and [25] (as in Lemma 2), each such computation is in NC. Combining the results into the linear combination is just one more parallel step. The only non-trivial work we have to do is in stage 1, and even there, step (c) is the bottleneck; all the other steps are easily seen to be in NC.

Stage 1 step (c) requires the computation of a perfect matching in the original graph G, which can be done in P. That is, perfect matchings can be counted provided we have our hands on one such matching. However, unlike [8], our ultimate goal is to search for a perfect matching (counting perfect matchings is a step toward this); we do not already know how to find one. One way around this would be as follows: "Add" a perfect matching to G_0 without violating planarity of G_0. To ensure that the generating function of perfect matchings does not change as a result, set $x'_e = 0$ for these newly added edges.

However, even such an addition is not exactly trivial. Let M be a pairing of the vertices of the graph and suppose that M is embedded within B_0 so that the embeddings of any pair of edges in M do not intersect. Note that an edge e in M may still intersect edges of G_0. However, we

may assume that each such intersection is a single point. Let $e_1, \ldots e_{n/2}$ denote the edges in M. We construct a sequence of graphs $G^0, G^1, \ldots, G^{n/2}$, where $G^0 = G$ and G^i is obtained from G^{i-1} essentially by adding e_i and introducing vertices at the points where e_i crosses edges in G^{i-1}.

More precisely, suppose that the edge $e_i = (a, b)$ is not an edge of G^{i-1} and crosses edges $f_j = (u_j, v_j)$ of G^{i-1}, for $1 \leq j \leq k$, in that order. (If e_i is already an edge in G^{i-1}, we simply set $G^i = G^{i-1}$.) We split each f_j into three edges $f_{j1} = (u_j, x_j)$, $f_{j2} = (x_j, y_j)$ and $f_{j3} = (y_j, v_j)$, by adding vertices x_j and y_j. We set $x'_{f_{j1}} = x_{f_j}$, and $x'_{f_{j2}} = x'_{f_{j3}} = 1$. Further, we add the edges (a, x_1), (y_j, x_{j+1}) for $1 \leq j \leq k-1$, and (y_k, b), and set $x'_e = 0$ for these edges. Figure 3 illustrates the setting when the x_e variables carry specific (fractional) values. Let us call these 0 weight edges the *edges corresponding to* e_i. Note that there is a monomial-preserving one-to-one correspondence between perfect matchings in G^{i-1} and perfect matchings in G^i that do not include any of the 0 weight edges corresponding to e_i. Indeed, from a perfect matching N in G^{i-1}, we obtain the corresponding perfect matching N' in G^i as follows: if (u_j, v_j) is present in N, we replace it by the edges (u_j, x_j) and (x_j, v_j); if (u_j, v_j) is not present in N, we add the edge (x_j, y_j) to N'. The existence of this correspondence shows that the generating functions of perfect matchings of G^{i-1} and G^i are the same.

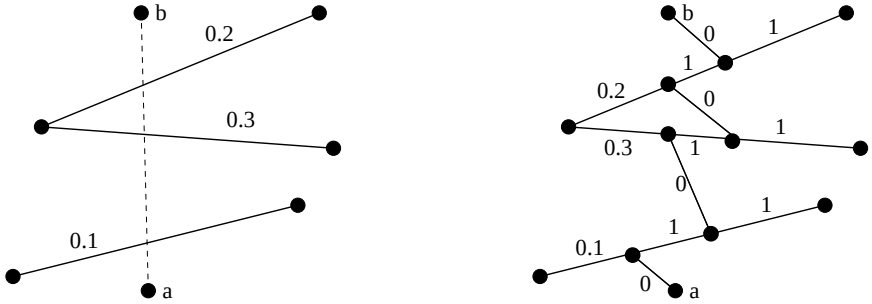

Figure 3: Adding edge (a, b) to transform G^{i-1} to G^i.

We conclude that $G^{n/2}$ has the same generating function as G. It also satisfies property 6 of Definition 8 – indeed the edges corresponding to edges in M form a perfect matching that lies within B_0. Finally, we note that although our definition of $G^{n/2}$ was sequential in that we "add" edges of M one after another, the construction can be easily parallelized because the edges of M do not intersect each other.

The only remaining point to address is obtaining the pairing M in NC. If a plane drawing of G were available, this would be trivial: sort all vertices by, say, x-coordinate (and break ties by y-coordinate), and pair up vertices in this ordering to obtain a non-crossing matching. What we actually have, however, is a planar combinatorial embedding, not a plane embedding, of G_0. Using the techniques of [31] (Theorems 5.3 and 5.4), we can retrieve, in NC, a plane drawing of G_0 if G_0 is biconnected. If G_0 is not biconnected, we can make it biconnected as follows: identify the cut-vertices of G_0. For each cut-vertex v, let its neighbors in cyclic order be $w_1, w_2, \ldots, w_k$. (If v is on the boundary, then assume that w_1 and w_k are the neighbors on the boundary.) Now introduce edges forming a path $w_1 w_2 \ldots w_k$ that connects all the neighbors, and set $x'_e = 0$ for these edges. This ensures that v is no longer a cut-vertex, while preserving planarity and the generating function of matchings.

Thus all stages of the algorithm of [8] can be parallelized, and we can conclude

Theorem 9 *Let G be a graph embedded on S_g given by its polygonal embedding as above. We can count perfect matchings of G in parallel time $O(g) + (\log n)^{O(1)}$, using $4^g n^{O(1)}$ processors. In particular, when $g \in O(\log n)$, the procedure is in NC.*

Using this in conjunction with Theorem 1, we have

Theorem 10 *Perfect matchings in bipartite graphs of logarithmic genus, given with a suitable embedding, can be found in NC.*
More generally, let G be a bipartite graph embedded on S_g given by its polygonal embedding as above. We can obtain a perfect matching of G (or report that none exists) in parallel time $O(g)+(\log n)^{O(1)}$, using $4^g n^{O(1)}$ processors.

5 A different search technique in small-genus bipartite graphs

In this section, we present a different algorithm for finding a perfect matching in a logarithmic genus bipartite graph. We first present the algorithm for planar bipartite graphs, then discuss the essential points of difference between this and the algorithm of Section 3, and then present the extension to small genus graphs.

We are given an embedding of a planar bipartite graph $G = (V, E)$, where V is the vertex set and E is the edge set. We assume that G has no loops or multi-edges.

We assume that the graph G is given to us as an embedding in some standard form.

5.1 The Algorithm for planar bipartite graphs

The algorithm is shown in Figure 4. We first compute a rational "flow" or "weight" $x_e \in [0, 1]$ on each edge $e \in E$ satisfying the following *conservation constraint*: for each vertex $v \in V$, the *demand* at v, defined to be the sum of the weights of edges incident to v, equals one. To do this, we simply set x_e to be the ratio of the number of perfect matchings in which e participates to the total number of perfect matchings. It is clear that this assignment satisfies the conservation constraint. As described in the summary, we can compute this assignment by using the counting algorithm as a subroutine.

Next, we manipulate the weight on the edges, always maintaining the conservation constraint, till we obtain an integral assignment of weights which corresponds to a perfect matching. In what follows, G will always denote the subgraph consisting of the edges with non-zero weights. When the weight on some edge becomes zero, it is deleted from G.

The manipulation of the weights is accomplished by the while loop. Using the algorithm in Lemma 4, we pick a subset of faces in G such that no two faces share an edge. The number of faces in this subset is a constant fraction of the total number of faces in G. For each face in parallel, we extract a simple cycle C from the edges bounding the face. (Note that such a simple cycle always exists.) Since G is bipartite, C has an even number of edges. Let e denote the edge on C with the smallest weight, and let x denote this weight. We decrease the weights of all edges of C at an even distance from e by x, and increase the weights of all edges of C at an odd distance from e by x. (Since C is an even cycle, "odd distance" (resp. "even distance") from e is well-defined.) We delete from G the edge e and any other edges of C whose weight becomes zero.

In the above step, we can operate on each cycle in parallel because the cycles are edge-disjoint. It is straightforward to check that at the end of the step the weight of any edge is in the range

Compute M = the number of perfect matchings in G.
If $M = 0$, exit.
For each $e \in E$, pardo:
    Compute M_e = the number of perfect matchings in $G - \{e\}$.
    If $M_e = M$ Then delete edge e
        Else Assign weight $1 - (M_e/M)$ to (a, b).
    Endif
while G has a cycle do
    Find a set S of edge-disjoint cycles in G.
    For each cycle $C \in S$, pardo:
        Find the smallest edge weight, say x on edge e.
        Decrease weight of all edges at even distance from e on C by x.
        Increase weight of all edges at odd distance from e on C by x.
        Delete all edges on C whose weight is zero.
endwhile

Figure 4: Algorithm $\mathcal{A}$ for finding a perfect matching in a planar bipartite graph

$(0, 1]$, and that the conservation constraint continues to hold. This completes the description of an iteration of the while loop, and also the algorithm.

The correctness of the algorithm is established below.

Lemma 3 *When the algorithm terminates, the edges of G form a perfect matching.*

Proof: It is clear that the algorithm terminates with an acyclic graph (i.e. forest, since the graph is undirected) G and a weight $x_e \in (0, 1]$ on edge e of G satisfying the conservation constraint. Let G' be any connected component of G. Due to the conservation constraint, G' is not an isolated vertex. Since G is acyclic, G' is a tree. Let v be any pendant vertex of G', and let (u, v) be the unique edge incident on v. Because the demand at v equals one, the weight of (u, v) must equal one. But since the demand at u does not exceed one, (u, v) is the only edge incident on u as well. Thus, G' consists of just the edge (u, v). It follows that the edge set of G forms a perfect matching. ■

We now show that the procedure is indeed implementable in NC.

Lemma 4 *In a planar graph G with f faces, a set of $\Omega(f)$ edge-disjoint faces and $\Omega(f)$ edge-disjoint simple cycles can be found in NC.*

Proof: A planar graph G as well as its dual G^* and pruned dual G^{**} are sparse graphs. So they do not have too many high degree vertices. In G^{**}, call a vertex high-degree if it has 12 or more neighbors. Since $e^{**} \leq 3n^{**} - 6 = 3f - 6$, there are less than $f/2$ high-degree vertices; the rest are all of low degree. Now consider the subgraph of G^{**} induced by these low degree vertices, and construct any maximal independent set S. For each vertex that we put in S, at most 11 others stay

out. So 1 out of every 12 low-degree vertices must be in S; i.e. $S \geq f/24$. And each independent set in G^{**} gives a set of edge-disjoint of faces in G.

To find the set of faces, use any NC algorithm for finding maximal independent sets in G^{**} (see [24] for general graphs, or [12] for planar graphs). If G^{**} has fewer than 24 faces, use a singleton set corresponding to any face as the independent set.

For each face F in the above set, consider the graph with only those edges that bound F. A spanning tree in this graph and a fundamental cycle with respect to it can be found in NC. And since the faces are edge-disjoint, so are these cycles. (If the edges bounding a face do not contain a cycle (i.e. they form a forest), then it must be the case that the entire graph has only one face.) ■

Lemma 5 *In $O(\log n)$ iterations of the while loop, G becomes acyclic.*

Proof: We argue that the number of iterations of the while loop is $\log |F|$, where F is the initial set of faces. It follows that the number of iterations is $O(\log n)$.

Since any simple cycle divides the plane into two faces, a one-face planar graph is acyclic. Thus it suffices to prove that after each iteration of the while loop, the number of faces in G falls by a constant fraction. This follows from the following two observations. At the beginning of the loop we pick a subset of edge-disjoint faces which constitute a constant fraction of the set of all faces. And at the end of the loop, we delete one edge from each picked face, thus destroying the face. ■

Theorem 11 *Given a planar bipartite graph G on n vertices, algorithm $\mathcal{A}$ (from Figure 4 either produces a perfect matching M in G, or reports that G has no perfect matching. $\mathcal{A}$ runs in time $(\log n)^{O(1)}$ using $n^{O(1)}$ processors.*

Proof: The initial assignment of weights satisfying the conservation constraint can be found in NC (use Lemma 2). Each iteration of the while loop can be implemented in NC. (Use Lemma 4 to find the set of faces. Killing each face is straightforward.) By Lemma 5, the number of iterations of the while loop is $O(\log n)$. ■

5.2 Comparison with other algorithms

In the preceding subsection, we described an NC algorithm to construct a perfect matching in a planar bipartite graph. There are three other algorithms for this problem: (1) the algorithm of Miller and Naor [27], (2) the algorithm from Section 3, where the part from Proposition 3 is implemented using Gabow's edge coloring algorithm [7], and (3) the algorithm from Section 3, where the part from Proposition 3 is implemented using the processor-efficient version of [9]. We comparing our algorithm with each of these.

It has been known for quite some time that computing an $s-t$ max-flow or min-cut in a planar graph is in NC. All algorithms for these problems use the correspondence between cuts in the graph and cycles in the dual graph. This is also true of the Miller-Naor algorithm for planar bipartite matching, where bipartite matching is viewed as the problem of computing a flow with multiple sources and sinks. (Note that the standard reduction of bipartite matching to $s-t$ max-flow does not maintain planarity.) This correspondence between cuts and cycles in the dual graph breaks down in graphs of higher genus. We are not aware of any NC algorithm for max-flow or min-cut

in such graphs. Our algorithm for planar bipartite matching conceptually quite different from the algorithm of Miller and Naor because it does not use this correspondence, and is thus amenable to higher-genus graphs as well.

In comparing our algorithm with (2) and (3) above, one notable difference that is immediately visible is the following: Our algorithm maintains the invariant that at each stage, we are within the perfect matching polytope $\mathcal{PM}(G)$. This is not true for the algorithms based on Proposition 3. These algorithms first round down the fractional matching $\vec{x}$ to integral multiples of $\alpha = 1/2^{\lceil \log m \rceil + 1}$. This process could take one outside $\mathcal{PM}(G)$, since the equality constraint is no longer satisfied at each vertex unless the initial point already had all edge weights multiples of α. Thus for the perfect matching case, these are not truly interior-point algorithms.

The crucial observation used in (2), (3) is that though outside $\mathcal{PM}(G)$, the rounded point is within $\mathcal{M}(G)$ and in fact "close to" $\mathcal{PM}(G)$. The algorithm of (2) considers the rounded weights y_e, and constructs a multigraph G' by replacing each edge e with Δy_e parallel edges, where $\Delta = 1/\alpha$. Then G' has maximum degree Δ, and can be edge-colored with Δ colors. The closeness of $\vec{y}$ to $\mathcal{PM}(G)$ guarantees that at least one color class is a perfect matching. This is completely different from what we do with the interior point.

The algorithm of (3) is closest in spirit to our algorithm. It manipulates edge weights while monotonically increasing closeness to $\mathcal{PM}(G)$. On the other hand, we manipulate edge weights while maintaining membership in $\mathcal{PM}(G)$, but monotonically decrease the complexity of the graph itself (Lemma 5). Also, the algorithm of (3) uses global weight information (the distribution of weights in an Euler trail) to decide how to manipulate the weights. Our algorithm uses global weight-independent information to find a maximal set of disjoint cycles, but subsequently each cycle is manipulated using weight information local to the cycle.

5.3 Extension to small genus graphs

Let G be a bipartite graph embedded on an orientable surface S_g of genus g. In this section, we describe how the algorithm of the previous section can be extended to find a perfect matching in G. The overall scheme is quite similar to the algorithm of Figure 4.

We are left with an acyclic graph with weights satisfying the conservation constraint. By the argument in Lemma 3, this must be a perfect matching. In the rest of this section, we briefly describe the NC implementations of each of the above steps when the graph genus is logarithmically bounded.

In each iteration in Step 2 of the algorithm, we find a subset of edge-disjoint faces of G. A subset whose cardinality is a constant fraction of the total number of faces of G can be found using the algorithm of the following lemma.

Lemma 6 *Let G be a graph embedded on S_g given by its polygonal embedding. We can find a subset of $\Omega(f)$ edge-disjoint faces in G in NC, where f is the number of faces in G, provided $f \geq 4g$.*

Proof: We construct the dual G^* of G, and delete from G^* any self-loops and multiple copies of any edge to get G^{**}. Let $n^{**} = f$, e^{**}, and f^{**} denote, respectively, the number of vertices, edges, and faces in G^{**}. Recall that G^{**} is a connected graph of genus at most g, and $e^{**} \leq 3n^{**} + 6(g-1)$.

Now, as in Lemma 4, we can argue that at least $n^{**}/2 - g$ vertices of G^{**} have small (less than 12) degree, and so a maximal independent set has size at least $n^{**}/24 - g/12$. Since $n^{**} = f \geq 4g$, this set has size at least $f/48$. And this independent set corresponds to a subset of edge-disjoint faces in G. ■

1. Using the counting algorithm as a subroutine, find a convex combination of perfect matchings of G, i.e. a weight on each edge so that the conservation constraint is satisfied at each vertex of G.

2. As long as the graph G has more than one face remaining, repeat the following. Find a subset of faces in G such that no two faces share an edge. For each picked face f in parallel, find a simple separating cycle C from the edges bounding the face. Manipulate the weight on the edges in C as before, and delete from G the edges with zero weight.

3. The graph G now has only one face, but since it is an embedding on a surface of higher genus, it might still have cycles. Repeat the following until the graph becomes acyclic. Pick any cycle in G, and manipulate the weights on the edges of the cycle until some edge-weight becomes zero. Delete the edges of C with zero weight from the graph.

Figure 5: Algorithm $\mathcal{B}$ for finding a perfect matching in a genus g bipartite graph

For each face in the subset, we find an appropriate cycle bounding part of the face using the following algorithm.

Lemma 7 *Let graph G be embedded on S_g, with more than one face, and let f be a face of the embedding. There exists a simple cycle C using edges incident to f with the following property: every edge of C is a separating edge. That is, if we delete any edge in C from G, the face f merges with some adjacent face different from f. Further, such a cycle can be found in NC.*

Proof: We look at the edges incident to f and throw away any edge that has f on both sides of it. It is easy to see that the remaining set of edges forms a union of simple cycles, and deleting any of these edges causes f to merge with some adjacent face. We pick any one of these cycles. ■

Lemma 8 *In $O(g + \log n)$ iterations of the while loop, G is reduced to a single face graph.*

Proof: The while loop has two phases: first, when there are at least $4g$ faces in the graph, and second, when the reduced graph has fewer than $4g$ faces.

In the first phase, we select, at the beginning of each iteration, a subset of edge-disjoint faces which constitute a constant fraction of the set of all faces. In the second phase, we pick any one face at the beginning of each iteration.

In either phase, at the end of the loop, we delete one edge from each picked face, such that the picked face then merges with an adjacent face (that was not picked).

We now argue that the number of iterations of the while loop is $O(\log |F| + g)$, where F is the initial set of faces. ($|F| \in O(n^2)$.) After each iteration of the while loop in the first phase, the number of faces in G falls by a constant fraction; clearly, there are at most $O(\log |F|)$ such iterations. And the second phase has no more than $4g$ iterations, since a face is destroyed in each iteration. ■

Lemma 9 *The number of iterations in Step 3 of the algorithm is at most* $2g$.

Proof: The analogue of Jordan's Curve Theorem on the plane states that if a surface has genus g, then any set of 2g+1 closed curves will make the surface disconnected. (See for instance [1].)

At the beginning of step 3, the graph G is embedded onto S_g and the embedding has a single face. Since any cycle in G will embed on to a closed curve on S_g, it follows that G has at most $2g$ cycles. And each iteration of step 3 destroys at least one cycle. ■

Theorem 12 *Given a bipartite graph G on n vertices along with its embedding on a polygonal representation of an oriented surface of genus g, algorithm $\mathcal{B}$ either produces a perfect matching M in G, or reports that G has no perfect matching.*

$\mathcal{B}$ runs in time $(\log n)^{O(1)}g$ using $4^g n^{O(1)}$ processors.

In particular, if $g \in O(\log n)$, then $\mathcal{B}$ is an NC algorithm.

Proof: The bounds for step 1 are stated in Theorem 9. Lemma 6 and Lemma 7 show that each iteration of step 2 and of step 3 is in NC. Lemma 8 and Lemma 9 bound the number of iterations for steps 2 and 3 respectively. ■

6 Finding a half-integral solution in a small-genus graph in NC

In this section we make partial progress towards finding a perfect matching in a planar graph in NC. Recall that $\mathcal{PM}(G) \subseteq \mathcal{FPM}(G)$, and that once the bipartiteness condition is relaxed, we no longer have $\mathcal{PM}(G) = \mathcal{FPM}(G)$. Our eventual goal is to find a vertex of $\mathcal{PM}(G)$. Here we show that we can find a vertex of $\mathcal{FPM}(G)$ in NC. Namely, we establish the following theorem.

Theorem 13 *For graphs of logarithmic genus, a vertex of the fractional matching polytope $\mathcal{FPM}(G)$ (i.e. a half-integral solution to the equations defining $\mathcal{FPM}(G)$, with no even cycles) can be found in NC, provided that the perfect matching polytope $\mathcal{M}(G)$ is non-empty.*

As in Section 5, we present the algorithm for the planar case. The extension to small genus is exactly as in Section 5.3.

Our starting point is the same interior point p computed in the previous section; namely, the arithmetic mean of all perfect matchings of G. Starting from p, we attempt to move towards a vertex. The basic strategy is to find a large set S of edge-disjoint faces. Each such face contains a simple cycle, which we try to destroy. Difficulties arise if the edges bounding the faces in S do not contain even length simple cycles, since the method of the previous section works only for even cycles. We describe mechanisms to be used successively in such cases.

6.1 Basic Building Blocks

We first describe some basic building blocks, and then describe how to put them together.

Building block 1: Simplify, or *Standardize*, the graph G.

Let G be the current graph, let $x : E \longrightarrow \mathcal{Q}$ be the current assignment of weights to edges, and let y be the partial assignment finalized so far. The final assignment is $y : E \longrightarrow \{0, 1/2, 1\}$.

Step 1.1 For each $e = (u, v) \in E(G)$, if $x_e = 0$, then set $y_e = 0$ and delete e from G.

Step 1.2 For each $e = (u, v) \in E(G)$, if $x_e = 1$, then set $y_e = 1$ and delete u and v from G.

(This step ensures that all vertices of G have degree at least 2.)

Step 1.3 Obtain connected components of G.

If a component is an odd cycle C, then every edge on C must have weight $1/2$. For each $e \in C$, set $y_e = 1/2$. Delete all the edges and vertices of C from G.

If a component is an even cycle C, then for some $0 < a < 1$, the edges on C alternately have weights a and $1 - a$. For each $e \in C$, if $x_e = a$ then set $y_e = 1$ and if $x_e = 1 - a$ then set $y_e = 0$. Delete all the edges and vertices of C from G.

Step 1.4 Let V' be the set of vertices of degree 2 in G. Consider the subgraph of G induced by V'; this is a disjoint collection of paths. Collapse each such even path to a path of length 2 and each such odd path to a path of length 1, reassigning weights as shown in Figure 6. Again, we stay within the polytope of the new graph, and from any assignment here, a point in $\mathcal{FPM}(G)$ can be recovered in a straightforward way.

This step ensures that no degree 2 vertex has a degree 2 neighbor.

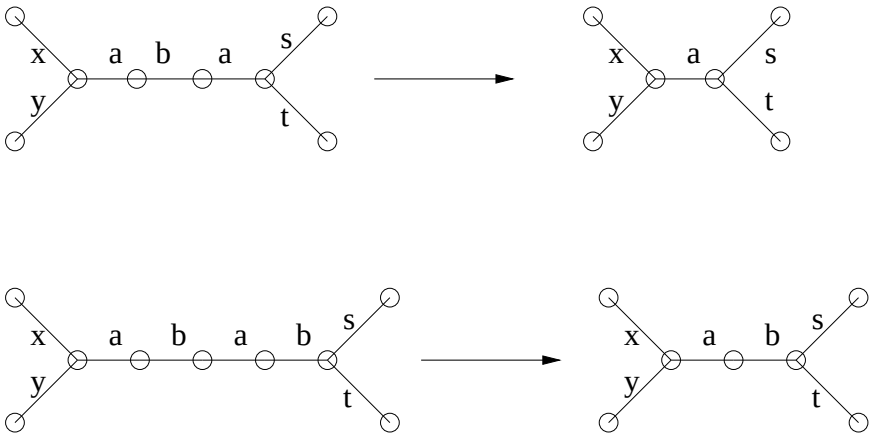

Figure 6: Transformation assuring that no two consecutive vertices are of degree 2

Step 1.5 For each $v \in V(G)$, if v has degree more than 3, then introduce some new vertices and edges, rearrange the edges touching v, and assign weights as shown in Figure 7. This assignment in the new graph is in the corresponding polytope of the new graph, and from any assignment here, a point in $\mathcal{FPM}(G)$ can be recovered in a straightforward way. (This gadget construction was in fact first used in [5].)

This step ensures that all vertices have degree 2 or 3.

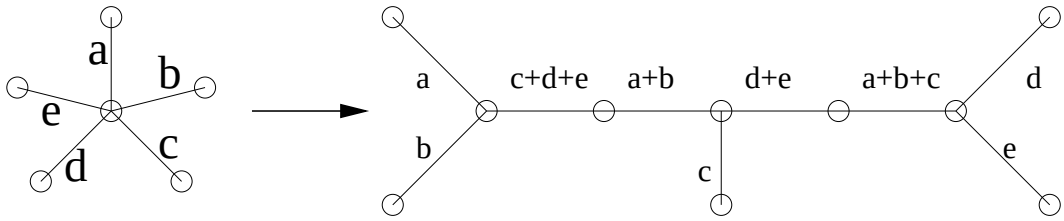

Figure 7: Transformation to remove vertices of degree greater than 3

Note that Steps 1.4 and 1.5 above change the underlying graph. To recover the point in $\mathcal{FPM}(G)$ from a point in the new graph's polytope, we can initially allocate one processor per edge. This processor will keep track of which edge in the modified graph dictates the assignment to this edge. Whenever any transformation is done on the graph, these processors update their respective data, so that recovery at the end is possible.

We will call a graph on which the transformations of building block 1 have been done a *standardized graph.*

Building Block 2: Process an even cycle. This is as in Section 5.

Building Block 3: Process an odd cycle connected to itself by a path. Let C be such an odd cycle, with path P connecting C to itself. We first consider the case when P is a single edge, i.e. a chord. The chord (u, v) cuts the cycle into paths P_1, P_2. Let C_i denote the cycle formed by P_i along with the chord (u, v). Exactly one of C_1, C_2 is even; process it as in Building Block 2.

If instead of a chord, there is some path $P_{u,v}$ connecting u and v on C, the same reasoning holds and so this step can still be performed.

Building Block 4: Process a pair of edge-disjoint odd cycles connected by a path.

Let C_1 and C_2 be the odd cycles and P the path connecting them. Note that if G is standardized, then P cannot be of length 0. Let P connect to C_1 at u and to C_2 at v. Then the traversal of C_1 beginning at u, followed by path P going from u to v, then the traversal of C_2 beginning at v, followed by the path P going from v to u, is a closed walk of even length. We make two copies of P, one for each direction of traversal. For edge e on P, assign weight $x_e/2$ to each copy. Now treating the two copies as separate, we have an even cycle which can be processed according to building Block 2. For each edge $e \in P$, its two copies are at even distance from each other, so either both increase or both decrease in weight. It can be seen that after this adjustment, the weights of the copies still adds up to something between 0 and 1.

This step is illustrated in Figure 8. The edge on the path has weight a is split into two copies with weight $a/2$ each. The dotted edge is the minimum weight edge; thus $w \leq a/2$.

6.2 The Algorithm

The idea is to repeatedly identify large sets of edge-disjoint faces, and then manipulate them, in the process destroying them. The faces are identified as in Section 5, and a simple cycle is extracted from each face. Even cycles are processed using Building Block 2. By the building blocks 3 and

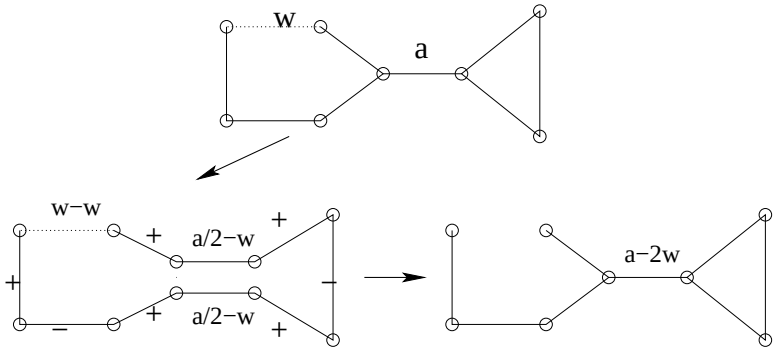

Figure 8: Manipulating a closed walk of even length

4, odd cycles can also be processed provided we identify paths connecting the cycles to themselves or other cycles. However, to achieve polylogarithmic time, we need to process several odd cycles simultaneously, and this requires that the odd cycles and the connecting paths be edge-disjoint.

We use the following definition: A path P is said to be a *3-bounded* path if the number of internal vertices of P with degree 3 is at most 1. Note that in a standardized graph, a 3-bounded path can have at most 3 internal vertices.

The algorithm can be described as follows:

1. Find a point $p \in \mathcal{M}(G)$ and initialize x_e accordingly.
2. Standardize G (Building block 1; this builds a partial solution represented in y).
3. While G is not empty, repeat the following steps, working in parallel on each connected component of G:
   (a) Find a collection S of edge-disjoint faces in G, including at least 1/24 of the faces from each component. Extract a simple cycle from the edges bounding each face of S, to obtain a collection of simple cycles T.
   (b) Process all even cycles (Building block 2). Remove these cycles from T. Re-standardize.
   (c) Define each surviving cycle in T to be a cluster. (At later stages, a cluster will be a set of vertices and the subgraph induced by this subset.)
   While T is non-empty, repeat the following steps:
      i. Construct an auxiliary graph H with clusters as vertices. H has an edge between clusters D_1 and D_2 if there is a 3-bounded path between some vertex of D_1 and some vertex of D_2 in G.
      ii. Process all clusters having a self-loop in H. (Building Block 3). Remove these clusters from T. Re-standardize.
      iii. Recompute H. In H, find a maximal matching. Each matched edge pairs two clusters, between which there is a 3-bounded path in G. In parallel, process all these pairs along with the connecting path using Building Block 4. Remove the processed clusters from T and re-standardize.

iv. "Grow" each cluster: if D is the set of vertices in a cluster, then first add to D all degree-2 neighbors of D, then add to D all degree-3 neighbors of D.

4. Return the edge weights stored in y.

6.3 Correctness

The correctness of the algorithm follows from the following series of lemmas:

Lemma 10 *The clusters and 3-bounded paths processed in Step 3(c) are vertex-disjoint.*

Proof: Each iteration of the while loop in Step 3(c) operates on different parts of G. We show that these parts are edge-disjoint, and in fact even vertex-disjoint. Clearly, this is true when we enter Step 3(c); the cycles are edge-disjoint by our choice in Step 3(a), and since G has maximum degree 3, no vertex can be on two edge-disjoint cycles. Changes happen only in steps 3(c)(ii) and 3(c)(iii); we analyze these separately.

Consider Step 3(c)(ii). If two clusters D_1 and D_2 have self-loops, then there are 3-bounded paths ρ_i from D_i to D_i, $i = 1, 2$. If these paths share a vertex v, it can only be an internal vertex of ρ_1 and ρ_2, since the clusters were vertex-disjoint before this step. In particular, v cannot be a degree-2 vertex. But since G is standardized, $\deg(v)$ is then 3, which does not allow it to participate in two such paths. So ρ_1 and ρ_2 must be vertex-disjoint. Thus processing them in parallel via building block 4 is valid. Processing clusters with self-loops merely removes them from T; thus the clusters surviving after Step 3(c)(ii) continue to be vertex-disjoint.

Now consider Step 3(c)(iii). Suppose cluster D_1 is matched to D_2 via 3-bounded path ρ, D_3 to D_4 via 3-bounded path η. Note that $D_i \neq D_j$ for $i \neq j$, since we are considering a matching in H. Thus, by the same argument as above, the paths ρ and η must be vertex-disjoint. Thus processing them in parallel via building block 4 is valid. Processing matched clusters removes them from T; the remaining clusters continue to be vertex-disjoint.

Since Step 3(c)(iii) considers a maximal matching, the clusters surviving are not only vertex-disjoint but also not connected to each other by any 3-bounded path. ■

Lemma 11 *Each invocation of the while loop inside Step 3(c) terminates in finite time.*

Proof: To establish this statement, we will show that clusters which survive Steps 3(c)(ii) and 3(c)(iii) grow appreciably in size. In particular, they double in each iteration of the while loop. Clearly, clusters cannot double indefinitely while remaining vertex-disjoint, so the statement follows. In fact, our proof establishes that the while loop in Step 3(c) executes $O(\log n)$ times on each invocation.

Let G denote the graph at the beginning of Step 3(c). Consider a cluster at this point. Let D_0 be the set of vertices in the cluster. Consider the induced subgraph $G[D_0]$ on D_0. Notice that each such $G[D_0]$ contains exactly one cycle, which is an odd cycle extracted in Step 3(a).

We trace the progress of cluster D_0. Let D_i denote the cluster (or the associated vertex set; we use this notation interchangeably to mean both) resulting from D_0 after i iterations of the while loop of Step 3(c). If D_0 does not survive i iterations, then D_i is empty.

For any cluster D, let 3-size(D) denote the number of vertices in D whose degree in G is 3. Let D' denote the vertices of D whose degree in G is 3 but degree in D is 1.

We establish the following claim:

Claim 14 *For a cluster D_0 surviving $i+1$ iterations of the while loop of Step 3(c), $G[D_i]$ contains exactly one cycle, and furthermore,*

$$\begin{aligned} |{D_i}'| &\geq \lfloor 2^{i-1} \rfloor \\ \textit{3-size}(D_i) &\geq 2^i \end{aligned}$$

Proof of Claim: As mentioned earlier, $G[D_0]$ contains exactly one cycle. Thus 3-size$(D_0') = 0$. In fact, each $G[D_j]$, $j \leq i$ contains just this one cycle, because if any other cycle were present in $G[D_j]$, then a self-loop would be found at the $(j+1)$th stage and the cluster would have been processed and deleted from T in Step 3(c)(ii); it would not grow $(i+1)$ times.

It remains to establish the claims on the sizes of D_i and D_i'. We establish these claims explicitly for $i \leq 1$, and by induction for $i > 1$.

Consider $i = 0$. Clearly, 3-size$(D_0) \geq 2^0 = 1$, and $\lfloor 2^{-1} \rfloor = 0$.

Now consider $i = 1$. We know that D_0 has gone through two "Grow" phases, and that $G[D_1]$ has only one cycle. Notice that each degree 3 vertex in D_0 contributes one vertex *outside* D_0; if its third non-cycle neighbor were also on the cycle, then the cycle has a chord detected in Step 3(c)(ii) and D_0 does not grow even once. In fact, since D_0 grows twice, the neighbors are not only outside the cycle but are distinct from each other. Thus for each vertex contributing to 3-size(D_0), one degree-3 vertex is added to D_1 and these vertices are distinct. Thus all these vertices are in D_1' giving $|D_1'| =$ 3-size$(D_0) \geq 1$, and 3-size$(D_1) =$ 3-size$(D_0) + |D_1'| \geq 2$.

To complete the induction, assume that the claim holds for $i-1$, where $i > 1$. In this case, $\lfloor 2^{i-2} \rfloor = 2^{i-2}$. Thus 3-size$(D_{i-1}) \geq 2^{i-1}$, and $|D_{i-1}'| \geq 2^{i-2}$.

Each $u \in D_{i-1}'$ has two neighbors, u_1 and u_2, not in D_{i-1}. These vertices contributed by each member of D_{i-1}' must be disjoint, since otherwise D_{i-1} would have a 3-bounded path to itself and would be processed at the ith stage; it would not grow the ith time. Furthermore, if u_l is of degree 2, let u_l' denote its degree-3 neighbor other than u; otherwise let $u_l' = u_l$. By the same reasoning, the vertices u_1', u_2' contributed by each $u \in D_{i-1}'$ must also be disjoint. So $2|D_{i-1}'|$ vertices are added to D_{i-1} in obtaining D_i. All these new vertices must be in D_i' as well, since otherwise D_i would have a 3-bounded path to itself and would be processed at the $(i+1)$th stage; it would not grow the $(i+1)$th time. Hence $|D_i'| = 2|D_{i-1}'| \geq 2 \cdot 2^{i-2} = 2^{i-1}$.

Every degree-3 vertex of D_{i-1} continues to be in D_i and contributes to 3-size(D_i). Furthermore, all the vertices of D_i' are not in D_{i-1} and also contribute to 3-size(D_i). Thus 3-size$(D_i) =$ 3-size$(D_{i-1}) + |D_i'| \geq 2^{i-1} + 2^{i-1} = 2^i$. ■ Now that the Claim has been proved, its geometric growth implies Lemma 11, and in fact implies that the "while" loop in Step 3(c) executes $O(\log n)$ times in each invocation. ■

Lemma 12 *The while loop of Step 3 terminates in finite time.*

Proof: Suppose some iteration of the while loop in Step 3 does not delete any edge. This means that Step 3(b) does nothing, so S has no even cycles, and Step 3(c) deletes nothing, so the clusters keep growing. But by the preceding claim, the clusters can grow at most $O(\log n)$ times; beyond that, either Step 3(c)(ii) or Step 3(c)(iii) must get executed.

Thus each iteration of the while loop of Step 3 deletes at least one edge from G, so the while loop terminates in finite time. ■

Lemma 13 *After step 2, and after each iteration of the while loop of Step 3, we have a point inside* $\mathcal{FPM}(G)$.

Proof: It is easy to see that all the building blocks described in Section 6.1 preserve membership in $\mathcal{FPM}(G)$. Hence the point obtained after Step 2 is clearly inside $\mathcal{FPM}(G)$. During Step 3, various edges are deleted by processing even closed walks. By our choice of S, the even cycles processed simultaneously in Step 3(b) are edge-disjoint. By Lemma 10, the even closed walks processed simultaneously in Steps 3(c)(ii) and 3(c)(iii) are edge-disjoint. Now all the processing involves applying one of the building blocks, and these blocks preserve membership in $\mathcal{FPM}(G)$ even if applied simultaneously to edge-disjoint even closed walks. The statement follows. ■

Lemma 14 *When the algorithm terminates, we have a vertex of* $\mathcal{FPM}(G)$.

Proof: When G is empty, all edges of the original graph have edge weights in the set $\{0, 1/2, 1\}$. Consider the graph H induced by the non-zero edge weights y_e. From the description of Building Block 1, it follows that H is a disjoint union of a partial matching (with edge weights 1) and odd cycles (with edge weights 1/2). Such a graph must be a vertex of $\mathcal{FPM}(G)$ (see, for instance, [23]). ■

6.4 Analysis

It is clear that each of the basic steps of the algorithm runs in NC. The proof of Lemma 11 establishes that the while loop inside Step 3(c) runs $O(\log n)$ times. To show that the overall algorithm is in NC, it thus suffices to establish the following:

Lemma 15 *The while loop of Step 3 runs* $O(\log n)$ *times.*

Proof: Let F be the maximum number of faces per component of G at the beginning of step 3. We show that F decreases by a constant fraction after each iteration of the while loop of step 3. Since $F = O(n)$ for planar graphs, it will follow that the while loop executes at most $O(\log n)$ times.

At the start of Step 3, the connected components of G are obtained, and they are all handled in parallel. Let us concentrate on any one component. Within each component, unless the component size (and hence the number of faces f in the embedding of this component) is very small, $O(1)$, a set of $\Omega(f)$ edge-disjoint faces (in fact, $f/24$ faces) and $\Omega(f)$ edge-disjoint simple cycles can be found in NC. This is established in Lemma 3 of [26], which basically shows that a maximal independent set amongst the low-degree vertices of the dual is the required set of faces.

So let T be the set of edge-disjoint faces obtained at the beginning of step 3. If $|T| \leq f/24$, then the component is very small, and it can be processed sequentially in $O(1)$ time. Otherwise, note that after one iteration of the while loop of Step 3, T is emptied out, so all the clusters in T get processed. A single processing step handles either one cluster (cluster with self-loop) or two (clusters matched in H), so at least $|T|/2$ processing steps are executed (not necessarily sequentially).

Each processing step deletes at least one edge. Let the number of edges deleted be $k \geq |T|/2$. Of these, k_1 are not bridges at the time when they are deleted and $k_2 = k - k_1$ are bridges when they are deleted. Each deletion of a non-bridge merges two faces in the graph. Thus if $k_1 \geq |T|/4$,

then at least $|T|/4 \geq f/96$ faces are deleted; the number of faces decreases by a constant fraction. If $k_2 > |T|/4$, consider the effect of these deletions. Each bridge deleted is on a path joining two clusters. Deleting it separates these two clusters into two different components. Thus after k_2 such deletions, we have at least k_2 pairs of separated clusters. Any connected component in the resulting graph has at most one cluster from each pair, and hence *does not have* at least k_2 clusters. Since each cluster contains an odd cycle and hence a face, the number of faces in the new component is at most $f - k_2 \leq f - |T|/4 \leq f - f/96$. Hence, either way, the new F is at most $95F/96$, establishing the lemma. ■

7 Conclusions

This paper presents an interior-point method based on counting to solve the search problem for perfect matchings in NC. The method works for those subclasses of bipartite graphs for which counting is in NC. A somewhat different and simpler method works for small genus bipartite graphs. In recent work [18], our approach has been extended to cubic and polylog degree bipartite graphs. For these graphs, too, searching for a perfect matching was already known to be in NC. But the algorithm of [18] is completely different from earlier algorithms and much closer in spirit to that of Section 5. This suggests that the essence of the algorithm does not hinge on topological constraints (small genus) but on interior-point manipulations.

The obvious question is how to extend this approach to non-bipartite graphs. We believe that counting is a harder problem than search; hence for graphs where counting perfect matchings is in NC, finding one also ought to be in NC. Our attempt at obtaining this generalization has worked only partially, and only for small genus graphs; here we can find a half-integral perfect matching (a vertex of $\mathcal{FPM}(G)$) in NC, provided there is at least one perfect matching to begin with. Unfortunately, we still do not know any way of navigating from a vertex of $\mathcal{FPM}(G)$ to a vertex of $\mathcal{PM}(G)$ in general. Proving this at least for planar graphs would be an important first step.

Note that for planar non-bipartite graphs, a sublinear time ($O(n^{1/2+\epsilon})$) algorithm can be easily devised using separators: find a separator S, match vertices of S sequentially to some unmatched neighbor, then recursively process the unmatched components.

Another open problem is to find embeddings of graphs onto surfaces of small genus in NC, or even to test whether such embeddings exist. No NC algorithms are known for testing or embedding onto surfaces of genus greater than 0. While testing whether a graph has genus at most k is an NP-complete problem [32], for a fixed k the problem is known to be in P and even in linear time [28].

A combinatorially interesting, though not necessarily practical, question is whether for planar graphs (or for that matter, any class of graphs where counting perfect matchings is in NC), one can count half-integral perfect matchings (vertices of $\mathcal{FPM}(G)$) in NC.

8 Acknowledgements

We thank Jan Vondrák for pointing us to the reference [8]. We thank Jaikumar Radhakrishnan for helping us with the transformation in the proof of Theorem 9. We thank Srikanth Iyengar and S Venkatesh for helpful discussions. We thank Laci Babai for suggesting looking at other interior-

point method algorithms in the matching context. We are grateful to the referees, whose comments helped us to significantly improve the presentation of our results.

References

[1] C. P. Bonnington and C. H. C. Little. *The foundations of topological graph theory.* Springer, 1995.

[2] M. Chrobak, J. Naor, and M. Novick. Using bounded degree spanning trees in the design of efficient algorithms on claw free graphs. In *Proceedings of the Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science Vol 382*, pages 147–162, Ottawa, Canada, 1989. Springer-Verlag.

[3] E. Dahlhaus, P. Hajnal, and M. Karpinski. On the parallel complexity of Hamiltonian cycles and matching problem in dense graphs. *Journal of Algorithms*, 15:367–384, 1993.

[4] E. Dahlhaus and M. Karpinski. The matching problem for strongly chordal graphs is in NC. Technical Report 855-CS, University of Bonn, 1986.

[5] E. Dahlhaus and M. Karpinski. Perfect matching for regular graphs is AC^0-hard for the general matching problem. *Journal of Computer and System Sciences*, 44(1):94–102, 1992.

[6] E. Dekel and S. Sahni. Parallel matching algorithm for convex bipartite graphs and applications to scheduling. *Journal of Parallel and Distributed Computing*, 1(2):185–205, 1984.

[7] H. N. Gabow. Using euler partitions to edge-colour bipartite multi-graphs. *International Journal of Computer and Information Science*, 5(4):345–355, Dec 1976.

[8] A. Galluccio and M. Loebl. On the theory of Pfaffian orientations I: Perfect matchings and permanents. *The Electronic Journal of Combinatorics*, R6, 1999.

[9] M. Goldberg, S. Plotkin, D. Shmoys, and É. Tardos. Using interior-point methods for fast parallel algorithms for bipartite matching and related problems. *SIAM Journal on Computing*, 21(1):140–150, 1992.

[10] M. Goldberg, S. Plotkin, and M. Vaidya. Sublinear time parallel algorithms for matching and related problems. *Journal of Algorithms*, 14:180–213, 1993.

[11] D. Grigoriev and M. Karpinski. The matching problem for bipartite graphs with polynomially bounded permanents is in NC. In *Proceedings of 28th IEEE Conference on Foundations of Computer Science*, pages 166–172. IEEE Computer Society Press, 1987.

[12] X. He. A nearly optimal parallel algo for constructing maximal independent set in planar graphs. *Theoretical Computer Science*, 61:33–47, 1988.

[13] J. Ja'Ja'. *Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[14] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6:35–48, 1986.

[15] M. Karpinski and W. Rytter. *Fast parallel algorithms for graph matching problems*. Oxford University Press, Oxford, 1998. Oxford Lecture Series in Mathematics and its Applications 9.

[16] P. W. Kastelyn. Graph theory and crystal physics. In F. Harary, editor, *Graph Theory and Theoretical Physics*, pages 43–110. Academic Press, 1967.

[17] D. Kozen, U. Vazirani, and V. Vazirani. NC algorithms for comparability graphs, interval graphs, and testing for unique perfect matching. In *Proceedings of FST&TCS Conference, LNCS Volume 206*, pages 496–503. Springer-Verlag, 1985.

[18] R. Kulkarni. A new NC algorithm for perfect matchings in regular bipartite graphs of polylog degree. In *Proceedings of CIAC Conference, LNCS Volume 3998*, pages 308–319. Springer-Verlag, 2006.

[19] R. Kulkarni and M. Mahajan. Seeking a vertex of the planar matching polytope in NC. In *Proceedings of the 12th European Symposium on Algorithms ESA, LNCS vol. 3221*, pages 472–483. Springer, 2004.

[20] G. Lev, M. Pippenger, and L. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C-30:93–100, 1981.

[21] C. H. C. Little. An extension of Kasteleyn's method of enumerating the 1-factors of planar graphs. In *Combinatorial Mathematics, Proc. 2^{nd} Australian Conference*, pages 63–72. D.Holton, *Lecture Notes in Mathematics*, 403, 1974.

[22] L. Lovász. On determinants, matchings and random algorithms. In L. Budach, editor, *Proceedings of Conference on Fundamentals of Computing Theory*, pages 565–574. Akademia-Verlag, 1979.

[23] L. Lovász and M. Plummer. *Matching Theory*. North-Holland, 1986. Annals of Discrete Mathematics 29.

[24] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.

[25] M. Mahajan, P. R. Subramanya, and V. Vinay. The combinatorial approach yields an NC algorithm for computing Pfaffians. *Discrete Applied Mathematics*, 143(1-3):1–16, September 2004.

[26] M. Mahajan and K. Varadarajan. A new NC-algorithm for finding a perfect matching in planar and bounded genus graphs. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pages 351–357, 2000.

[27] G. Miller and J. Naor. Flow in planar graphs with multiple sources and sinks. *SIAM Journal on Computing*, 24:1002–1017, 1995.

[28] B. Mohar. A linear time algorithm for embedding graphs in an arbitrary surface. *SIAM Journal on Discrete Mathematics*, 12(1):6–26, Feb 1999.

[29] K. Mulmuley, U. Vazirani, and V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–131, 1987.

[30] I. Parfenoff. An efficient parallel algorithm for maximum matching for some classes of g raphs. *Journal of Parallel and Distributed Computing*, 52(1):96–108, 1998.

[31] R. Tamassia and J. F. Vitter. Parallel transitive closure and point location in planar structures. *SIAM Journal on Computing*, 20(4):708–725, 1991.

[32] C. Thomassen. The graph genus problem is NP-complete. *Journal of Algorithms*, 10(4):568–576, Dec 1989.

[33] C. Thomassen. *Embeddings and minors*, pages 301–349. Elsevier Science, 1995.

[34] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[35] V. Vazirani. NC algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems. *Information and Computation*, 80(2):152–164, 1989.

[36] V. Vazirani. Parallel graph matching. In J. Reif, editor, *Synthesis of Parallel Algorithms*. Kaufman-Morgan, 1991. chapter 18.

[37] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York Inc., 1999.

[38] A. T. White. *Graphs, Groups and Surfaces*. North-Holland, Amsterdam, 1973.