Chapter 6

Fast NC and RNC Algorithms for Finding a Maximal Set of Paths

6.1 Parallel algorithms for the MPS problem

In this section, we first present an RNC algorithm for the MPS problem for undirected graphs. We then mention how to modify it for digraphs. At the end of this section, we will also point out how to derandomize them, and apply the resulting algorithms to approximating the shortest superstring problem.

A vertex v in G is *good* if $\sum_{u \in N_G(v)} \frac{1}{d_G(u)} \geq \frac{1}{3}$. An edge in G is *good* if at least one of its endpoints is good. The following two lemmas will be used in our algorithms:

Lemma 6.1 [II86, Koz92] At least half the edges in every graph G are good.

Lemma 6.2 [II86] There is an RNC algorithm which, given a graph $G = (V, E)$, computes a matching M in G such that for each good vertex v in G, the probability that $v \in V(M)$ is no less than a positive constant. It runs in $O(1)$ expected time with $O(n + m)$ processors on a CRCW PRAM.

6.1.1 Description of the RNC algorithm

We first give an intuitive explanation of the algorithm. The algorithm always maintains a set F of edges in G such that the graph (V, F) is a collection of vertex-disjoint paths. Initially, $F = \emptyset$. The algorithm then proceeds in stages. Before each stage gets started, every edge e in G satisfies that $F \cup \{e\}$ is a path set. In each stage, the algorithm uses Lemma 6.2 to compute a matching M between the endpoints of the paths in the graph (V, F). It is clear that the graph $(V, F \cup M)$ is a collection of vertex-disjoint paths and cycles. To break the cycles, for each path P in the graph (V, F), the algorithm randomly chooses one of the two endpoints of P, and deletes from M the edge incident to the unchosen endpoint (if it really exists). Obviously, the resulting M satisfies that each path in the graph $(V, F \cup M)$ contains at most one edge in M. This implies that $F \cup M$ is also a path set. The algorithm then adds the edges in M to F, and deletes from G every edge e such that $F \cup \{e\}$ is not a path set. After this, it proceeds to the next stage if G still contains at least one edge; otherwise, it outputs F and halts.

To implement each stage efficiently, the algorithm maintains an array R for which the following is an *invariant*: For each vertex $i \in V$, $R[i] = i$ if the degree of i in the graph (V, F) is 0, $R[i] = -1$ if the degree of i in the graph (V, F) is 2, and $R[i] = j$ otherwise, where j is the other endpoint

of the path containing i in the graph (V, F). Initially, the algorithm sets $R[i] = i$ for each vertex $i \in V$. Thus, the invariant is true before the first stage gets started.

More precisely, the algorithm performs the following steps in each stage:

1. Use the algorithm in Lemma 6.2 to find a random matching M' in G.
2. For each $i \in V(M')$, select i if $R[i] = i$, and randomly select one of i and $R[i]$ if $i < R[i]$. [*Comment:* $R[i] \neq -1$ for every vertex $i \in V(M')$.]
3. Set M to be the set of those edges $e \in M'$ such that both endpoints of e were selected in step 2.
4. Add the edges in M to F.
5. In parallel, for each edge $\{i_0, i_1\} \in M$, perform the following steps:
   - 5.1. For each $i_j \in \{i_0, i_1\}$, if $R[i_j] \neq i_j$ (or equivalently, the degree of i_j in the graph (V, F) is now 2), then remove all the edges incident to i_j from G, and set $R[i_j] := -1$.
   - 5.2. Remove the edge $\{R[i_0], R[i_1]\}$ from G if it is in G (since the graph $(V, F \cup \{\{R[i_0], R[i_1]\}\})$ contains a cycle).
   - 5.3. Set $R[R[i_0]] := R[i_1]$ and $R[R[i_1]] := R[i_0]$.

Clearly, every edge e such that $F \cup \{e\}$ is not a path set must be deleted either in step 5.1 or 5.2, and the array R is updated in steps 5.1 and 5.3 so that the invariant remains true. By a simple induction on the number of stages the algorithm performs, we can show the correctness of the algorithm.

6.1.2 Complexity analysis

In this subsection, we prove the following theorem:

Theorem 6.3 The RNC algorithm runs in $O(\log n)$ expected time using $O(n + m)$ processors on a CRCW PRAM.

The algorithm uses $O(n + m)$ processors; every vertex and every edge in G has a processor associated with it. Clearly, the initialization step of the algorithm takes $O(1)$ time with $O(n)$ processors on an EREW PRAM. We claim that each stage of the algorithm can be done in $O(1)$ time with $O(n + m)$ processors on a CRCW PRAM. According to Lemma 6.2, step 1 of each stage can be done in $O(1)$ time with $O(n + m)$ processors on a CRCW PRAM. Other steps in each stage use no more resources. This establishes the claim. In the remainder of this subsection, we will show that the expected number of stages performed by the algorithm is $O(\log n)$. This together with the claim implies the theorem.

First, we show the following lemma:

Lemma 6.4 Fix a stage of the algorithm. For each good vertex i in G, the probability that $i \in V(M)$ at the end of the stage is no less than a positive constant. Consequently, at the end of the stage, $\Pr[i \in V(M) \text{ or } j \in V(M)]$ is no less than a positive constant for each good edge $\{i, j\}$ in G.

Proof. By Lemma 6.2, for each good vertex i in G, the probability that $i \in V(M')$ after step 1 of the stage is no less than a positive constant. It is not difficult to see that $\Pr[i \in V(M) \mid i \in V(M')] \geq \frac{1}{4}$ for all vertices i in G. Thus, we have the lemma. ∎

We proceed to the proof of the fact that the expected number of stages performed by the algorithm is $O(\log n)$. To show the fact, we use a potential function argument. For a subgraph $\mathcal{G}$ of the input graph and a path set $\mathcal{F}$, define

$$\Phi(\mathcal{G}, \mathcal{F}) = \sum_{\text{edge } \{i,j\} \text{ in } \mathcal{G}} \phi(\mathcal{G}, \mathcal{F}, \{i, j\}),$$

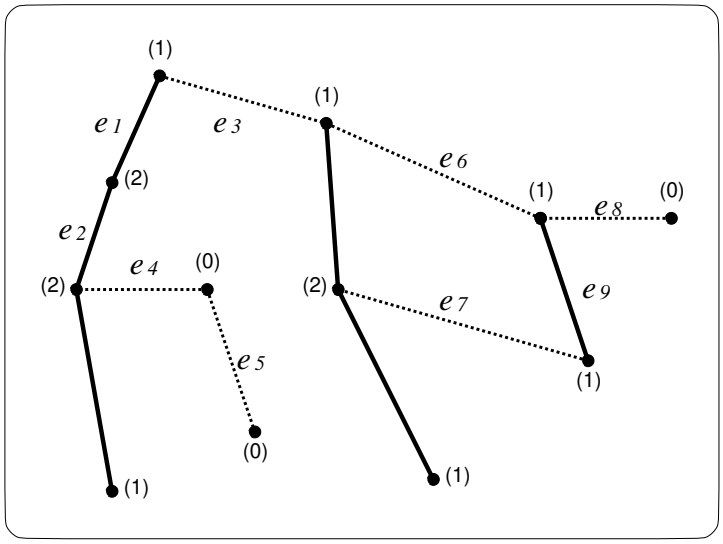

Figure 6.1: Example of the potential function for an undirected graph

where $\phi(\mathcal{G}, \mathcal{F}, \{i, j\}) = (2 - d_{(V,\mathcal{F})}(i))(2 - d_{(V,\mathcal{F})}(j))$.

Example 6.5 Let G be a graph given in Figure 6.1, and F be the set of the thick lines. The number near by each vertex is its degree in the graph (V, F). Then, $\phi(G, F, e_5) = 4$, $\phi(G, F, e_8) = 2$, $\phi(G, F, e_3) = \phi(G, F, e_6) = \phi(G, F, e_9) = 1$, and $\phi(G, F, e_1) = \phi(G, F, e_2) = \phi(G, F, e_4) = \phi(G, F, e_7) = 0$. Thus, $\Phi(G, F) = 9$.

Fix a stage of the algorithm. Let G_b and G_a, respectively, be the graph G before and after the stage. Similarly, let F_b and F_a, respectively, be the path set F before and after the stage. Let $M = F_a - F_b$. Note that M is the matching computed at step 3 of the stage. Clearly, $\Phi(G_b, F_b) \geq \Phi(G_b, F_a) \geq \Phi(G_a, F_a)$. The intuition behind the potential function can be seen from Lemma 6.1, Lemma 6.2, and the following fact:

Fact 6.6 Assume that $e = \{i, j\}$ is an edge in G_b such that $i \in V(M)$ or $j \in V(M)$. Then, $\phi(G_b, F_b, e) - \phi(G_b, F_a, e) \geq 1$.

Proof. Let d_i and d_j be the degrees of i and j in the graph (V, F_b), respectively. According to the values of d_i and d_j, we distinguish four cases as follows:

Case 1: $d_i = d_j = 0$. Then, we have $\phi(G_b, F_b, e) = 4$. If both $i \in V(M)$ and $j \in V(M)$, then $\phi(G_b, F_a, e) = 1$; otherwise, $\phi(G_b, F_a, e) = 2$. Thus, $\phi(G_b, F_b, e) - \phi(G_b, F_a, e) \geq 2$.

Case 2: $d_i = 0$ and $d_j = 1$. Then, we have $\phi(G_b, F_b, e) = 2$. If $j \in V(M)$, then $\phi(G_b, F_a, e) = 0$; otherwise, $i \in V(M)$ and $\phi(G_b, F_a, e) = 1$. Thus, $\phi(G_b, F_b, e) - \phi(G_b, F_a, e) \geq 1$.

Case 3: $d_i = 1$ and $d_j = 0$. This case is similar to Case 2.

Case 4: $d_i = d_j = 1$. Then, we have $\phi(G_b, F_b, e) = 1$ and $\phi(G_b, F_a, e) = 0$. Thus, $\phi(G_b, F_b, e) - \phi(G_b, F_a, e) = 1$. ∎

For a random variable X, let $\mathcal{E}X$ denote the expected value of X, and let $\mathcal{E}(X \mid B)$ denote the expected value of X given that event B occurs.

Lemma 6.7 (Main Lemma.) $\mathcal{E}(\Phi(G_b, F_b) - \Phi(G_a, F_a)) \geq \mathcal{E}(\Phi(G_b, F_b) - \Phi(G_b, F_a)) \geq c \cdot \Phi(G_b, F_b)$ for some constant $c > 0$.

Proof. For each edge $e = \{i, j\}$ in G_b, let $X_e = \phi(G_b, F_b, e)$, $Y_e = \phi(G_b, F_a, e)$, $Z_e = X_e - Y_e$, and B_e be the event that $i \in V(M)$ or $j \in V(M)$. Let the number of edges in G_b be m_b. Clearly, $\Phi(G_b, F_b) \leq 4m_b$.

Fix an edge $e = \{i, j\}$ in G_b. By Fact 6.6, $\mathcal{E}(Z_e \mid B_e) \geq 1$. From this, it follows that $\mathcal{E}Z_e \geq \mathcal{E}(Z_e \mid B_e)\Pr[B_e] \geq \Pr[B_e]$. Thus, if e is good, then by Lemma 6.4, $\mathcal{E}Z_e \geq \Pr[B_e] \geq c'$ for some constant $c' > 0$. Combining this with the fact that G_a is a subgraph of G_b, we now have

$$\begin{aligned}
\mathcal{E}(\Phi(G_b, F_b) - \Phi(G_a, F_a)) &\geq \mathcal{E}(\Phi(G_b, F_b) - \Phi(G_b, F_a)) \\
&= \sum_{\text{edge } e \text{ in } G_b} \mathcal{E}Z_e \geq \sum_{\text{good edge } e \text{ in } G_b} \mathcal{E}Z_e \\
&\geq \sum_{\text{good edge } e \text{ in } G_b} \Pr[B_e] \geq \sum_{\text{good edge } e \text{ in } G_b} c' \geq \frac{c' m_b}{2}.
\end{aligned}$$

The last inequality follows from Lemma 6.1. On the other hand, we have $\Phi(G_b, F_b) \leq 4m_b$. Thus, $\mathcal{E}(\Phi(G_b, F_b) - \Phi(G_a, F_a)) \geq c' m_b/2 \geq \frac{c'}{8}\Phi(G_b, F_b)$. This completes the proof. ∎

Note that $\Phi(G, \emptyset) = 4m$ and that the algorithm halts when $\Phi(G, F) < 1$ (or equivalently, $\Phi(G, F) = 0$). Thus, by Lemma 6.7 above and Theorem 1.3 in [MR95], we immediately have that the expected number of stages performed by the algorithm is at most $\int_1^{4m} \frac{1}{cx} dx = O(\log n)$. This completes the proof of the theorem.

6.1.3 Extension to digraphs

Our algorithm for computing an MDPS in D always maintains a set B of arcs in D such that the digraph (V, B) is a collection of vertex-disjoint paths. Initially, $B = \emptyset$. The algorithm then proceeds in stages. Before each stage gets started, every arc e in G satisfies that $B \cup \{e\}$ is an MPS. In each stage, the algorithm uses Lemma 6.2 to compute a suitable matching M' in the underlying graph of D, and then uses M' to compute a set M of arcs in D such that no two arcs in M have a common endpoint. The remaining task of each stage of the algorithm is essentially the same as that of each stage of the algorithm in the subsection 6.1.1. The algorithm proceeds to the next stage if D still contains at least one arc; otherwise, it outputs B and halts.

To implement each stage efficiently, the algorithm maintains an array R for which the following is an *invariant*: For each vertex $i \in V$, $R[i] = i$ if the total degree of i in the digraph (V, B) is 0, $R[i] = -1$ if the total degree of i in the digraph (V, B) is 2, and $R[i] = j$ otherwise, where j is the other endpoint of the path containing i in the digraph (V, B). Initially, the algorithm sets $R[i] = i$ for each vertex $i \in V$.

More precisely, the algorithm performs the following steps in each stage:

1. Use the algorithm in Lemma 6.2 to find a random matching M' in the underlying graph of D.
2. For each $i \in V(M')$, select i if $R[i] = i$, and randomly select one of i and $R[i]$ if $i < R[i]$. [*Comment:* $R[i] \neq -1$ for every vertex $i \in V(M')$.]
3. Set $M := \{(i, j) \in A(D) \mid \{i, j\} \in M'\}$
   $-\{(i, j) \in A(D) \mid \{i, j\} \in M', (j, i) \in A(D), \text{ and } i > j\}$.
4. Add the edges in M to B.
5. In parallel, for each arc $(i_0, i_1) \in M$, perform the following steps:
   5.1. For each $i_j \in \{i_0, i_1\}$, if $R[i_j] \neq i_j$ (or equivalently, the total degree of i_j in the digraph (V, B) is now 2), then remove all the arcs incident to i_j from D, and set $R[i_j] := -1$.

5.2. Remove the arc $(R[i_1], R[i_0])$ from D if it is in D
(since the digraph $(V, B \cup \{(R[i_1], R[i_0])\})$ contains a cycle).
5.3. Remove all the arcs with tail i_0 or head i_1 from D.
5.4. Set $R[R[i_0]] := R[i_1]$ and $R[R[i_1]] := R[i_0]$.

We say that a vertex i in D is *good* if it is good in G (the underlying graph of D), and say that an arc in D is *good* if its tail or head is good.

Lemma 6.8 At least one third of the arcs in D are good.

Proof. Let E_1 be the set of those edges $\{i, j\}$ in G such that exactly one of (i, j) and (j, i) is in D', and let $E_2 = E(G) - E_1$. Let x_1 (or x_2) be the number of edges $e \in E_1$ (respectively, $e \in E_2$) such that e is good in G. Then, $x_1 + x_2 \geq \frac{|E_1|+|E_2|}{2}$ by Lemma 6.1. Moreover, D has $|E_1| + 2|E_2|$ arcs among which $x_1 + 2x_2$ are good. If $|E_1| \geq |E_2|$, then $x_1 + 2x_2 \geq \frac{|E_1|+|E_2|}{2} = \frac{|E_1|+2|E_2|}{2} - \frac{|E_2|}{2} \geq \frac{|E_1|+2|E_2|}{2} - \frac{|E_1|+2|E_2|}{6} \geq \frac{|E_1|+2|E_2|}{3}$; otherwise, $x_1 + 2x_2 = 2(x_1 + x_2) - x_1 \geq (|E_1| + |E_2|) - |E_1| = |E_2| \geq \frac{|E_1|+2|E_2|}{3}$. This completes the proof. ∎

From Lemma 6.4, it is easy to see the following lemma:

Lemma 6.9 For every stage of the algorithm, the set M computed at step 3 in the stage satisfies that $\Pr[i \in V(M)$ or $j \in V(M)]$ is no less than a positive constant for each good arc (i, j) in D.

To prove that the expected number of stages performed by the algorithm is $O(\log n)$, we need to modify the potential function in Section 6.1.2 as follows: For a subgraph $\mathcal{D}$ of the input digraph and a DPS $\mathcal{B}$, define

$$\Phi'(\mathcal{D}, \mathcal{B}) = \sum_{\text{arc } (i,j) \text{ in } \mathcal{D}} \phi'(\mathcal{D}, \mathcal{B}, (i, j)),$$

where $\phi'(\mathcal{D}, \mathcal{B}, (i, j)) = (1 - d^+_{D[\mathcal{B}]}(i))(1 - d^-_{D[\mathcal{B}]}(j))(2 - d_{D[\mathcal{B}]}(i))(2 - d_{D[\mathcal{B}]}(j))$.

Example 6.10 Let D be a digraph given in Figure 6.2, and B be the set of the bold arcs. The two numbers in the pair near by each vertex are its indegree and outdegree in the digraph (V, B), respectively. The underlying graph of D is the same as the graph G in Figure 6.1, and a_i corresponds to e_i in Figure 6.1. Then, $\phi'(D, B, a_5) = 4$, $\phi'(D, B, a_6) = \phi'(D, B, a_9) = 1$, and $\phi'(D, B, a_1) = \phi'(D, B, a_2) = \phi'(D, B, a_4) = \phi'(D, B, a_7) = \phi'(D, B, a_3) = \phi'(D, B, a_8) = 0$. Thus, $\Phi'(D, B) = 6$.

Using Lemma 6.8 and Lemma 6.9, we can show the following lemma by a similar proof to that of Lemma 6.7:

Lemma 6.11 Fix a stage of the algorithm. Let D_b and D_a, respectively, be the digraph D before and after the stage. Similarly, let B_b and B_a, respectively, be the DPS B before and after the stage. Then, $\mathcal{E}(\Phi'(D_b, B_b) - \Phi'(D_a, B_a)) \geq c' \cdot \Phi'(D_b, B_b)$ for some constant $c' > 0$.

Using the same arguments as in the subsection 6.1.2, we get the following theorem.

Theorem 6.12 An MDPS can be computed in $O(\log n)$ expected time with $O(n+m)$ processors on a CRCW PRAM.

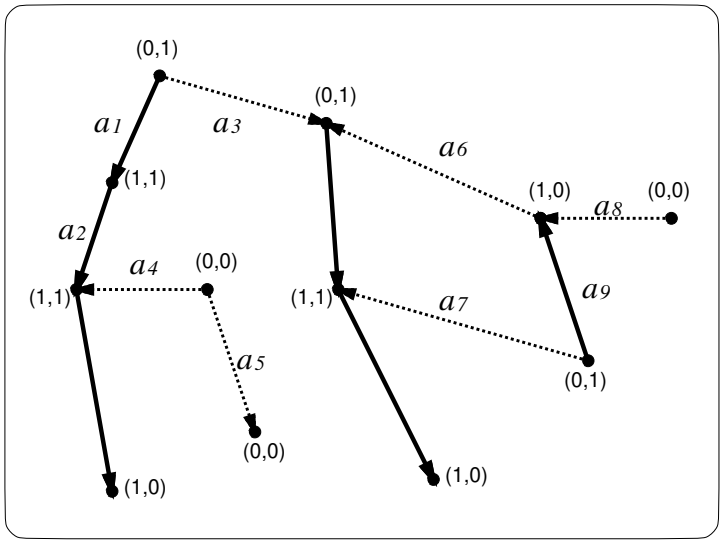

Figure 6.2: Example of the potential function for a directed graph

6.1.4 The NC algorithm

Carefully derandomizing the RNC algorithm in the subsection 6.1.1 with the techniques in [Che95a], we can obtain an NC algorithm for the MPS problem. We describe this result in the following theorem:

Theorem 6.13 An MPS in a given undirected graph can be found in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM.

Similarly, we can prove the following theorem:

Theorem 6.14 An MDPS in a given digraph can be found in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM.

It is not difficult to modify the proof of Theorem 6.14 to obtain a proof for the following corollary:

Corollary 6.15 Given a digraph D and a DPS F in D, an MDPS B in D with $F \subseteq B$ can be found in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM.

An immediate consequence of Corollary 6.15 is that the NC approximation algorithm for the shortest superstring problem given in [Che96] can be made faster. We describe this result in the following theorem.

Theorem 6.16 There is an NC approximation algorithm for the shortest superstring problem with a compression ratio of $\frac{1}{3+\epsilon}$ for any $\epsilon > 0$. It runs in $O(\log^2 n \cdot \log_{1+\epsilon/3}|S|)$ time with $\frac{n^2|S|^2}{\log n}$ processors, where n is the number of the given strings, and $|S|$ is the total length of the given strings.

6.2 An application to shortest superstrings with flipping

For a string s, let s^R denote the reverse of s and $|s|$ denote the length of s. Let s and t be two distinct strings, and let v be the longest string such that $s = uv$ and $t = vw$ for some non-empty strings u and w. $|v|$ is called the *overlap* between s and t and is denoted by $ov(s,t)$. By $s \circ t$, we denote the string uvw.

Let $S = \{s_1, s_2, \cdots, s_n\}$ be a set of strings over some alphabet Σ. Define $S^R = \{s_1^R, ..., s_n^R\}$ and $|S| = \sum_{i=1}^{n} |s_i|$. A *superstring-with-flipping* of S is a string s over Σ such that for each i, at least one of s_i and s_i^R is a substring of s. In the sequel, a superstring-with-flipping is simply called a superstring; this should be distinguished with the usual definition of a superstring in the literature. The *amount of compression* achieved by a superstring s is $|S| - |s|$. Let $opt_{com}(S)$ denote the maximum amount of compression achievable by a superstring of S. W.l.o.g., we assume that the set $S \cup S^R$ is *substring free*, i.e., no string in $S \cup S^R$ is a substring of any other. For simplicity of explanation, we assume that no string in S is a palindrome. At the end of this section, we will point out why this assumption is not essential to our results.

The *overlap graph* of S is a weighted digraph $OG(S) = (S \cup S^R, A, ov)$, where $A = \{(s,t) \mid s, t \in S \cup S^R,\ s \neq t$, and $s \neq t^R\}$ and each arc (s,t) has weight $ov(s,t)$. For a subgraph D of $OG(S)$, the *weight* of D is the total weight of the arcs in D and is denoted by $ov(D)$. The *mate* of a vertex s in $OG(S)$ is s^R. Similarly, the *mate* of an arc $e = (s,t)$ in $OG(S)$ is (t^R, s^R) and is denoted by e^R. Note that e and e^R have the same weight. A (directed) path P in $OG(S)$ is said to be *legal* if for every string $s \in S$, at most one of s and its mate s^R is on P. The *mate* of a legal path P in $OG(S)$ is the path consists of the mates of the arcs on P, and is denoted by P^R. Note that $ov(P) = ov(P^R)$. For a legal path $P = s, t, ..., u$ in $OG(S)$, we call $s \circ t \circ \cdots \circ u$ the *string associated with* P. Note that the string associated with P is a superstring of the strings s, t, $\cdots$, u and has length $(|s| + |t| + \cdots + |u|) - ov(P)$. A legal path P in $OG(S)$ is said to be *Hamiltonian* if for each string s in S, either s or its mate s^R is on P. A *two-path cover* of $OG(S)$ is a subgraph consisting of a Hamiltonian legal path and its mate. We denote by $opt_{cov}(S)$ the weight of a maximum-weight two-path cover of $OG(S)$. Then, we have the following fact immediately:

Fact 6.17 $opt_{cov}(S) = 2 \cdot opt_{com}(S)$. Moreover, the amount of compression achieved by the string associated with a Hamiltonian legal path P in $OG(S)$ is $ov(P)$. Especially, the amount of compression achieved by the string associated with a maximum-weight Hamiltonian legal path is $opt_{com}(S)$.

Recall that a directed path set (DPS) in a digraph $D = (V_D, A_D)$ is a subset B of A_D such that the digraph (V_D, B) is an acyclic digraph in which the indegree and outdegree of each vertex are both at most 1. Consider the following simple algorithm for finding a two-path cover of $OG(S)$ with large weight:

Algorithm GREEDY
Input: $OG(S) = (S \cup S^R, A, ov)$.
1. Initialize B to be the empty set.
2. While the digraph $(S \cup S^R, B)$ is not a two-path cover of $OG(S)$, perform the following: Add to B the maximum-weight arc e and its mate e^R such that $B \cup \{e, e^R\}$ is a DPS in $OG(S)$ but $B \cup \{f, f^R\}$ is not a DPS in $OG(S)$ for all arcs f with $ov(e) < ov(f)$. (Note: We here assume that the arcs of equal weights in $OG(S)$ are linearly ordered.)
3. Output the two-path cover $(S \cup S^R, B)$.

The following fact was implicitly mentioned in [JLD92]:

Fact 6.18 [JLD92] Let P be one of the two paths in the two-path cover output by GREEDY. Then, the amount of compression achieved by the string associated with P is at least $\frac{opt_{com}(S)}{2}$.

Lemma 6.19 Suppose that the weights on the arcs in $OG(S)$ are modified in a manner such that each arc has the same weight as its mate. Let $OG'(S) = (S \cup S^R, A, w)$ be the resulting digraph. Then, the two-path cover $(S \cup S^R, B)$ output by GREEDY on input $OG'(S)$ has weight $\geq \frac{w(C_{max})}{3}$, where C_{max} is a maximum-weight two-path cover of $OG'(S)$.

Proof. The proof uses similar ideas to those in [Tur89] and [CGPR94]. First observe that on input $OG'(S)$, GREEDY selects an arc and its mate during each iteration of the while-loop. Also note that the while-loop is executed exactly $n-1$ times. For $1 \leq i \leq n-1$, let e_i and e_i^R be the two arcs selected at the i-th iteration. For $0 \leq i \leq n-1$, let C_i be a maximum-weight two-path cover of $OG'(S)$ in which all the arcs e_1, e_1^R, e_2, e_2^R, $\cdots$, e_i, e_i^R appear. Clearly, C_0 is a maximum-weight two-path cover of $OG'(S)$ and C_{n-1} is the output of GREEDY on input $OG'(S)$.

We claim that for each $1 \leq i \leq n-1$, $w(C_{i-1}) \leq w(C_i) + 4w(e_i)$. To see the claim, fix an i with $1 \leq i \leq n-1$. Let $e_i = (s,t)$ and P be one of the two Hamiltonian legal paths in C_{i-1}. Then, $e_i^R = (t^R, s^R)$ and P^R is the other Hamiltonian legal path in C_{i-1}. Moreover, one of the following two cases must occur:

Case 1: s and t appear in the same Hamiltonian legal path in C_{i-1}. W.l.o.g., we may assume that s and t appear in P. Then, s^R and t^R appear in P^R. We only consider the case where s is reachable from t in P, t has a predecessor u in P, and s has a successor v in P; the other cases are similar. Then, there must exist an arc $(u_1, v_1) \notin \{e_1, e_1^R, ..., e_i, e_i^R\}$ on the subpath of P from t to s (possibly, $u_1 = t$ and/or $v_1 = s$). Moreover, the arcs (u,t), (u_1,v_1), (s,v) and their mates are contained in C_{i-1}. Let C be the two-path cover of $OG'(S)$ obtained from C_{i-1} by replacing the arcs (u,t), (u_1,v_1), (s,v) and their mates with the following six arcs: e_i, (u,v_1), (u_1,v) and their mates. Since the arcs (u,t), (u_1,v_1), (s,v) and their mates each have lighter weight than e_i by the algorithm GREEDY, we have $w(C) \geq w(C_{i-1}) - 4w(e_i)$. (*Note*: Recall that $w(e_i) = w(e_i^R)$.) On the other hand, since C is a two-path cover of $OG'(S)$ in which all the arcs e_1, e_1^R, e_2, e_2^R, ..., e_i, e_i^R appear, we have $w(C) \leq w(C_i)$ by the definition of C_i. Hence, we have $w(C_{i-1}) \leq w(C) + 4w(e_i) \leq w(C_i) + 4w(e_i)$.

Case 2: s and t do not appear in the same Hamiltonian legal path in C_{i-1}. W.l.o.g., we may assume that s appears in P and t appears in P^R. Then, s^R appears in P^R and t^R appears in P. We only consider the case where t^R is reachable from s in P and both s and t^R have a successor in P; the other cases are similar. Let u be the successor of s and v be the successor of t^R in P (possibly, $u = t^R$). Then, u^R and v^R must be the predecessors of s^R and t in P^R, respectively. Moreover, the arcs (s,u), (t^R,v) and their mates are contained in C_{i-1}. Let C be the two-path cover of $OG'(S)$ obtained from C_{i-1} by replacing the arcs (s,u), (t^R,v) and their mates with the following four arcs: e_i, (u^R,v) and their mates. Since the arcs (s,u), (t^R,v) and their mates each have lighter weight than e_i by the algorithm GREEDY, we have $w(C) \geq w(C_{i-1}) - 2w(e_i)$. (*Note*: Recall that $w(e_i) = w(e_i^R)$.) On the other hand, since C is a two-path cover of $OG'(S)$ in which all the arcs e_1, e_1^R, e_2, e_2^R, ..., e_i, e_i^R appear, we have $w(C) \leq w(C_i)$ by the definition of C_i. Hence, we have $w(C_{i-1}) \leq w(C) + 2w(e_i) \leq w(C_i) + 2w(e_i)$.

By the claim, we have $w(C_0) \leq w(C_{n-1}) + \sum_{i=1}^{n-1} 4w(e_i)$. Note that $w(C_0) = w(C_{max})$ and $w(C_{n-1}) = \sum_{i=1}^{n-1} 2w(e_i)$. Therefore, $w(C_{n-1}) \geq w(C_{max})/3$. This completes the proof. ∎

Our next goal is to parallelize GREEDY. To reach this goal, we need to prove several lemmas. First, several definitions are in order. An unweighted subgraph D of $OG(S)$ is said to be *legal* if the mate of each vertex and each arc in D is also contained in D. A DPS B in D is said

to be *legal* if the mate of each arc in B is also contained in B. A *maximal* legal DPS in D is a legal DPS that is not properly contained in another legal DPS. We want to design a parallel algorithm for computing a maximal legal DPS in a legal unweighted subgraph D of $OG(S)$. To this end, we modify the RNC algorithm in subsection 6.1.3 as follows. The new input is a legal unweighted subgraph D of $OG(S)$ and step 3 is replaced with the following three lines:

3'.1. For each $s_i \in S$, randomly select one of s_i and its mate;
3'.2. Set $M'' := \{e \in M' |$ both the tail and head of e were selected in the last step $\}$;
3'.3. Set $M := M'' \cup \{e |\ e$ is in D' and e^R is in $M''\}$;

The crucial point is that for each vertex (resp., arc) in D', the vertex (resp., arc) and its mate must be removed from D' in the same call of step 5. Moreover, from Lemma 6.9, it is easy to see the following lemma:

Lemma 6.20 For each good arc (i, j) in D', $\Pr[i \in V(M'')$ or $j \in V(M'')]$ is no less than a positive constant.

By this lemma and the discussions in subsection 6.1.3, we have:

Lemma 6.21 Given a legal unweighted subgraph D of $OG(S)$ and a legal DPS F in D, a maximal legal DPS B in D with $F \subseteq B$ can be found in $O(\log n)$ expected time with $O(n+m)$ processors on a CRCW PRAM, where m is the number of arcs in D.

To decrease the number of random bits used in line 3'.1 above, we further replace lines above with the following three lines:

3".1. Use M' to construct an undirected graph $K = (M', E_K)$, where E_K consists of those edges $\{e_1, e_2\}$ such that the head or tail of e_1 is the mate of the head or tail of e_2; (Comment: The maximum degree of K is 2.)
3".2. 3-color the vertices of K to partition M' into three independent sets M'_1, M'_2, and M'_3;
3".3. Randomly set M to be one of M'_1, M'_2, and M'_3;

It is easy to construct K and 3-color it in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM. Moreover, even if lines 3'.1 through 3'.3 are replaced with lines 3".1 through 3".3, Lemma 6.20 still holds. This together with the discussions in section 6.1 implies the following lemma:

Lemma 6.22 Given a legal unweighted subgraph D of $OG(S)$ and a legal DPS F in D, a maximal legal DPS B in D with $F \subseteq B$ can be found in $O(\log^2 n)$ time with $O(n^2(n+m)/\log n)$ processors on an EREW PRAM, where m is the number of arcs in D.

Now, we are ready to present a parallelized version of GREEDY. This algorithm is very similar to Algorithm 5 in [Che96].

Algorithm PARGREEDY
Input: $OG(S) = (S \cup S^R, A, ov)$.
1. Let $c = 1 + \frac{\epsilon}{3}$. In parallel, for each arc $e \in A$, set $lev(e) = \lceil \log_c ov(e) \rceil$ if $ov(e) > 1$, and set $lev(e) = 0$ otherwise.
2. Compute $MaxLev = \max\{lev(e) \mid e \in A\}$.
3. Set $B = \emptyset$ and $CurLev = MaxLev$.
4. While $CurLev \geq 0$, perform the following steps:
   4.1. Construct an unweighted digraph $D = (S \cup S^R, E)$ by setting $E = B \cup \{e \in A \mid lev(e) = CurLev\}$. (Comment: Since $ov(e) = ov(e^R)$ for each arc $e \in A$, D must be a legal unweighted subgraph of $OG(S)$.)

4.2 Compute a maximal legal DPS F in D with $B \subseteq F$ and then update B to be F.
4.3 Decrease $CurLev$ by 1.

5. Output the digraph $(S \cup S^R, B)$.

Lemma 6.23 Algorithm PARGREEDY finds a two-path cover of $OG(S)$ with weight at least $\frac{opt_{cov}(S)}{3+\epsilon}$.

Proof. The proof is very similar to that of Lemma 23 in [Che96]. Let C_{out} be the two-path cover output by PARGREEDY. Define a weighted digraph $OG'(S) = (S \cup S^R, A, w)$ by setting $w(e) = c^{lev(e)}$. Let $\succ$ be a total order on A satisfying the following condition: If either $w(e) > w(e')$, or $w(e) = w(e')$ and e is contained in C_{out} but e' is not contained in C_{out}, then $e \succ e'$. When $e \succ e'$, we say that e is *larger* than e'.

Let W be the maximum weight of a two-path cover of $OG'(S)$. Obviously, if the arcs in $OG'(S)$ are sorted in nonincreasing order using $\succ$, then given $OG'(S)$ as input to GREEDY (in place of $OG(S)$), GREEDY will output C_{out}. Thus, by Lemma 6.19, $w(C_{out}) \geq \frac{W}{3}$. On the other hand, $w(C_{out}) < c \cdot ov(C_{out})$ because $w(e) < c \cdot ov(e)$ for all $e \in A$. Therefore, $c \cdot ov(C_{out}) > \frac{W}{3} \geq \frac{opt_{cov}(S)}{3}$ since $w(e) \geq ov(e)$ for all $e \in A$. Recall that $c = 1 + \frac{\epsilon}{3}$. We now have $ov(C_{out}) > \frac{opt_{cov}(S)}{3c} = \frac{opt_{cov}(S)}{3+\epsilon}$. ∎

Theorem 6.24 There is a parallel approximation algorithm that produces a superstring of a given set S of n strings by which the amount of compression achieved is at least $\frac{1}{3+\epsilon}$ optimal for any $\epsilon > 0$. It runs in $O(\log n \cdot \log_{1+\epsilon/3} |S|)$ expected time with $O(|S|^2)$ processors on a CRCW PRAM or in $O(\log^2 n \cdot \log_{1+\epsilon/3} |S|)$ (deterministic) time with $O(|S|^2 + n^4/\log n)$ processors on an EREW PRAM.

Proof. To find a superstring of a given set S of n strings, we first construct $OG(S)$, next invoke the algorithm PARGREEDY to find a two-path cover C_{out} in $OG(S)$ with weight at least $\frac{opt_{cov}(S)}{3+\epsilon}$, and finally output the string associated with one of the two Hamiltonian legal path in C_{out}. By Fact 6.17 and Lemma 6.23, the amount of compression achieved by the output superstring is at least $\frac{1}{3+\epsilon}$ optimal.

As pointed out in [Che96], $OG(S)$ can be constructed from S in $O(\log |S|)$ time with $|S|^2$ processors on an EREW PRAM. Let us analyze the complexity of PARGREEDY. Clearly, steps 1, 2, and 3 can be done in $O(\log n)$ time with n^2 processors on an EREW PRAM. Noting that $MaxLev = O(\log_{1+\epsilon/3} |S|)$, we see that the while-loop is executed $O(\log_{1+\epsilon/3} |S|)$ times. Since steps 4.1 and 4.3 use no more resources than step 4.2, step 4 can be done in $O(\log n \cdot \log_{1+\epsilon/3} |S|)$ expected time with $O(n^2)$ processors on a CRCW PRAM by Corollary 6.21 or in $O(\log^2 n \cdot \log_{1+\epsilon/3} |S|)$ time using $O(n^4/\log n)$ processors on a CRCW PRAM by Corollary 6.22. The string associated with one of the two Hamiltonian legal path in C_{out} can be computed in $O(\log |S|)$ time with $|S|$ processors on an EREW PRAM. Thus, we can compute a superstring of S in $O(\log n \cdot \log_{1+\epsilon/3} |S|)$ expected time with $O(|S|^2)$ processors on a CRCW PRAM or in $O(\log^2 n \cdot \log_{1+\epsilon/3} |S|)$ (deterministic) time with $O(|S|^2 + n^4/\log n)$ processors on an EREW PRAM. ∎

Finally, we explain how to remove the assumption that no string in S is a palindrome. Suppose some strings in S are palindromes. We redefine $OG(S)$ as follows. For each string s in S, we introduce two vertices one of which corresponds to s and the other corresponds to s^R. That is, we treat s and s^R as different *vertices* in $OG(S)$, even if s is a palindrome. The edges of $OG(S)$ and their weights are defined as before. Redefining $OG(S)$ in this way, we give no influence on the above discussions.

6.3 Parallel algorithms for the MLF problem

6.3.1 Reducing the MLF problem to a simpler one

We first show a reduction of the MLF problem to a simpler one. To reduce the problem, we need additional definitions. A graph $G = (V, E)$ is said to be *proper* if V can be partitioned into two subsets X and Y such that for every $y \in Y$, (a) $X \cup \{y\}$ is an LFS in G and (b) $|X \cap N_G(y)| = 2$. To explicitly show this partition of V, we write $G = (X \cup Y, E)$. Hereafter, by a proper graph $G = (X \cup Y, E)$, we always mean that for every $y \in Y$, $X \cup \{y\}$ is an LFS in G and $|X \cap N_G(y)| = 2$. A *valid* MLFS in a proper graph $G = (X \cup Y, E)$ is an MLFS U in G with $X \subseteq U$. The *restricted* MLF problem is to find, given a proper graph $G = (X \cup Y, E)$, a valid MLFS in G. In the rest of this section, we describe an NC reduction from the MLF problem to the restricted MLF problem.

Algorithm *REDUCE*
Input: A graph $G = (V, E)$ with n vertices.
Output: A proper graph $H = (X \cup Y, E_H)$ such that $X \cup Y \subseteq V$, $H = G[X \cup Y]$, and every valid MLFS in H is an MLFS in G.
1. Initialize X to be an MIS in G.
2. Perform the following steps twice:
   2.1. Compute W, the set of all $v \in V - X$ such that $X \cup \{v\}$ is an LFS in G and $|N_G(v) \cap X| = 1$.
   2.2. Construct a graph $G' = (W, E_{G'})$, where $E_{G'}$ consists of all $\{v_1, v_2\}$ such that $\{v_1, v_2\} \in E$ or $N_G(v_1) \cap N_G(v_2) \cap X \neq \emptyset$.
   2.3. Compute an MIS I in G'.
   2.4. Add the vertices in I to X.
3. Set Y to be the set of those $v \in V - X$ such that $X \cup \{v\}$ is an LFS in G and $|N_G(v) \cap X| = 2$.

Let X_0 be the MIS in G found at step 1. For $i = 1, 2$, let W_i, G'_i, I_i, and X_i be the contents of the variables W, G', I, and X at the end of the ith execution of the steps 2.1 through 2.4, respectively.

Lemma 6.25 X_2 is an LFS in G.

Proof. X_0 is an MIS in G and is hence an LFS in G. We next prove that X_1 is an LFS in G. First note that $X_1 = X_0 \cup I_1$. By steps 2.1 through 2.3, each $v \in I_1$ has degree 1 in the graph $G[X_1]$. This together with the acyclicness of $G[X_0]$ implies that $G[X_1]$ is acyclic. We claim that every $u \in X_0$ has degree at most 2 in the graph $G[X_1]$. This can be seen by noting that each $u \in X_0$ is adjacent to at most one vertex $v \in I_1$ in the graph $G[X_1]$ and that $X_0 \cup \{v\}$ is an LFS in G for all $v \in I_1$. The claim implies that X_1 is an LFS in G. A similar discussion shows that X_2 is also an LFS in G. ∎

Lemma 6.26 For all $v \in V - X_2$, $X_2 \cup \{v\}$ is not an LFS in G or $|N_G(v) \cap X_2| \geq 2$.

Proof. Since X_0 is an MIS in G, it holds that $W_2 \subseteq W_1$. From this, we can observe that $V - X_2$ is the union of $V - (W_1 \cup X_0)$, $W_1 - (W_2 \cup I_1)$, and $W_2 - I_2$ which are pairwise disjoint. Let v be an arbitrary vertex in $V - X_2$. One of the following three cases must occur.

Case 1: $v \in V - (W_1 \cup X_0)$. Then, by the definition of W_1, $X_0 \cup \{v\}$ is not an LFS in G or $|N_G(v) \cap X_0| \geq 2$. Since $X_0 \subseteq X_2$, it follows that $X_2 \cup \{v\}$ is not an LFS in G or $|N_G(v) \cap X_2| \geq 2$.

Case 2: $v \in W_1 - (W_2 \cup I_1)$. Then, by the definition of W_2, $X_1 \cup \{v\}$ is not an LFS in G or $|N_G(v) \cap X_1| \geq 2$. Since $X_1 \subseteq X_2$, it follows that $X_2 \cup \{v\}$ is not an LFS in G or $|N_G(v) \cap X_2| \geq 2$.

Case 3: $v \in W_2 - I_2$. Since $v \in W_1$, $|N_G(v) \cap X_0| = 1$. Let u be the unique vertex in $N_G(v) \cap X_0$. Because $v \in W_1 - I_1$ and I_1 is an MIS in G'_1, there is a $w \in I_1$ with $\{v, w\} \in E_{G'_1}$. By the construction of G'_1, $\{v, w\}$ belongs to E or there is some $x \in X_0$ with $x \in N_G(v) \cap N_G(w)$. In the former case, $w \in N_G(v) \cap I_1$ and $\{u, w\} \subseteq N_G(v) \cap X_1$, implying that $|N_G(v) \cap X_2| \geq 2$. Thus, we may assume the latter case. Then, we have $x = u$ since $x \in X_0 \cap N_G(v) = \{u\}$. Therefore, $w \in N_{G[X_1]}(u)$. On the other hand, since $v \in W_2 - I_2$ and I_2 is an MIS in G'_2, there is a $w' \in I_2$ with $\{v, w'\} \in E_{G'_2}$. By the construction of G'_2, $\{v, w'\}$ belongs to E or there is some $y \in X_1$ with $y \in N_G(v) \cap N_G(w')$. In the former case, $w' \in N_G(v) \cap I_2$ and $\{u, w'\} \subseteq N_G(v) \cap X_2$, implying that $|N_G(v) \cap X_2| \geq 2$. Thus, we may assume the latter case. Then, we have $y = u$ since $|N_G(v) \cap X_1| = 1$ and $\{u, y\} \subseteq X_1 \cap N_G(v)$. Therefore, $w' \in N_{G[X_2]}(u)$. Recalling that $w \in N_{G[X_1]}(u)$, we now have $\{w, w'\} \subseteq N_{G[X_2]}(u)$. Hence, $\{v, w, w'\} \subseteq N_{G[X_2 \cup \{v\}]}(u)$. Obviously, $v \neq w$ and $v \neq w'$. Since $w \in I_1$, $w' \in I_2$, and $I_1 \cap I_2 = \emptyset$, it follows that $w \neq w'$. Therefore, $|N_{G[X_2 \cup \{v\}]}(u)| \geq 3$. This implies that $X_2 \cup \{v\}$ is not an LFS in G. ∎

Theorem 6.27 The MLF problem can be reduced to the restricted MLF problem in $O(\log n)$ expected time with n^2 processors on a CRCW PRAM, in $O(\log^2 n)$ time with n^4 processors on an EREW PRAM, or in $O(\log^3 n)$ time with n^2 processors on an EREW PRAM.

Proof. By Lemma 6.25 and Lemma 6.26, Algorithm *REDUCE* is clearly correct. This gives a reduction from the MLF problem to the restricted MLF problem. It remains to analyze the complexity of Algorithm *REDUCE*. Steps 1 and 2.3 can be done in $O(\log n)$ expected time with n^2 processor on a CRCW PRAM [Lub86], in $O(\log^2 n)$ time with n^4 processor on an EREW PRAM [Lub86], or in $O(\log^3 n)$ time with n^2 processor on an EREW PRAM [GS89]. It is not so difficult to see that the other steps can be done in $O(\log n)$ time using n^2 processors on an EREW PRAM. This establishes the theorem. ∎

In the next subsections, we present parallel algorithms for solving the restricted MLF problem.

6.3.2 Description of the RNC algorithm

The top-level structure of the RNC algorithm is as follows.

Algorithm *FOREST*
Input: A proper graph $G = (X \cup Y, E)$.
Output: A valid MLFS U in G.
1. Set $U := X$ and construct a new graph $G' = (Y, E')$, where E' is the set of all edges $\{y_1, y_2\}$ such that $\{y_1, y_2\} \in E$ or $X \cap N_G(y_1) \cap N_G(y_2) \neq \emptyset$.
2. While G' is not empty, perform the following steps:
   2.1. Select an independent set I in G' such that each connected component of $G[U \cup I]$ contains at most one vertex of I.
   2.2. Add the vertices in I to U.
   2.3. Delete from G' all vertices y such that $y \in I$ or $U \cup \{y\}$ is not an LFS in G.
3. Output U.

An easy induction shows that Algorithm *FOREST* correctly computes a valid MLFS of G. We need to specify how to implement the steps 2.1 and 2.3. In order to perform the two steps fast, the algorithm maintains an array R for which the following is an *invariant*: For each vertex $u \in U$, $R[u] = u$ if $\mathrm{d}_{G[U]}(u) = 0$, $R[u] = v$ if $\mathrm{d}_{G[U]}(u) = 1$, where v is the other vertex of degree 1 in the connected component of $G[U]$ containing u. (*Comment:* Vertices u in U with $\mathrm{d}_{G[U]}(u) = 2$ are irrelevant to R.) It is necessary to insert a new step between steps 1 and 2 in Algorithm

FOREST to initialize R so that the invariant is true before the first execution of the while-loop. Since the details are trivial, we omit them.

To implement step 2.1, we need a definition and a lemma due to Luby [Lub86]. Following [Koz92], we say that a vertex v in a simple graph H is *good* if $\sum_{u \in N_H(v)} \frac{1}{\mathrm{d}_H(u)} \geq \frac{1}{3}$. The following lemma is known:

Lemma 6.28 [Lub86, Koz92] Given a graph H, we can compute an independent set J in $O(1)$ expected time with $n+m$ processors on a CRCW PRAM such that for each good vertex v in H, the probability that $v \in N_H(J)$ is no less than a positive constant.

To implement step 2.1, we use the following three substeps:

2.1.1. Compute an independent set J of G' as in Lemma 6.28.

2.1.2. In parallel, for each vertex $u \in U$ with $\mathrm{d}_{G[U]}(u) \leq 1$, select u if $u = R[u]$, and randomly select one of u and $R[u]$ if $u < R[u]$.

2.1.3. Set $I := \{y \in J \mid$ the two vertices in $U \cap N_G(y)$ are both selected at step 2.1.2$\}$.

Lemma 6.29 For the set I computed in step 2.1.3, each connected component of $G[U \cup I]$ contains at most one vertex of I.

Proof. Fix an arbitrary connected component C of $G[U]$. Recall that C is a path. By Algorithm FOREST, only the endpoint(s) of C can be adjacent to a vertex of I in G. Moreover, if C has two endpoints, then exactly one of the two is selected in step 2.1.2. Thus, by the definition of I, at most one vertex u of C can be adjacent to a vertex of I in G, and this vertex u must be an endpoint of C. To show the lemma, it suffices to show that $|N_G(u) \cap I| = 1$. Towards a contradiction, assume that $|N_G(u) \cap I| \geq 2$. Let y_1 and y_2 be two vertices in $N_G(u) \cap I$. Note that either $u \in X$ or $u \in U - X$. In the former case, $\{y_1, y_2\}$ must be an edge in G' by step 1, contradicting that I is an independent set in G'. In the latter case, u must be in Y and hence $|N_G(u) \cap X| = 2$. Combining this with the fact that $X \subseteq U$, we see that u can not be an endpoint of C, a contradiction. This completes the proof. ∎

To update R and to implement step 2.3, we use the following substeps:

2.3.1. In parallel, for each $y \in I$, find the two vertices u_1 and u_2 in $N_G(y) \cap U$ and set $R[R[u_1]] := R[u_2]$ and $R[R[u_2]] := R[u_1]$.

2.3.2. Delete all vertices in I from G'.

2.3.3. In parallel, for each $u \in U$ with $\mathrm{d}_{G[U]}(u) = 2$, delete from G' all vertices in $N_G(u)$.

2.3.4. In parallel, for each vertex y in G', if the two vertices u_1 and u_2 in $N_G(y) \cap U$ satisfies that $R[u_1] = u_2$, then delete y from G'.

We claim that R is correctly updated in step 2.3.1. This simply follows from the fact that the two vertices in $N_G(y) \cap U$ both have degree at most 1 in $G[U]$.

Lemma 6.30 The vertices of G' deleted in step 2.3.3 or 2.3.4 are exactly those vertices y in G' such that $U \cup \{y\}$ is not an LFS in G.

Proof. It is clear that the vertices y of G' deleted in step 2.3.3 or 2.3.4 satisfy that $U \cup \{y\}$ is not an LFS in G. Let y be a vertex such that $U \cup \{y\}$ is not an LFS in G. Then, one of the following three cases must occur; (1) there is a vertex $u \in N_G(y) \cap U$ with $\mathrm{d}_{G[U \cup \{y\}]}(u) > 2$; (2) the two vertices in $N_G(y) \cap U$ are the two endpoints of a single path in $G[U]$; and (3) $\mathrm{d}_{G[U \cup \{y\}]}(y) > 2$. In the cases (1) and (2), y is deleted in step 2.3.3 and 2.3.4, respectively. We claim that the case (3) is impossible. Towards a contradiction, assume that the case (3) occurs. Then, since $|N_G(y) \cap X| = 2$, there must exist a vertex $u \in (U - X) \cap N_G(y)$. Since $U - X \subseteq Y$, u was

added to U in an earlier iteration of the while-loop in Algorithm *FOREST*. In that iteration, y must have been deleted in step 2.3.3 because $\mathrm{d}_{G[U]}(u) = 2$. This contradicts that y still remains in G' in the current iteration. ∎

6.3.3 Complexity Analysis

Suppose that the while-loop in Algorithm *FOREST* is executed t times. For $1 \leq i \leq t$, let U_i and G'_i be respectively the contents of the variables U and G' after the ith iteration, and let I_i and J_i be the two independent sets (in G'_i) computed in the ith iteration. For convenience, let U_0 and G'_0 be the contents of the variables U and G' before the first iteration. Our main task in this subsection is to prove an upper bound on the expected value of t. We begin with a helpful lemma.

Lemma 6.31 For every $0 \leq i \leq t-1$ and every good vertex v in G'_i, $\Pr[v \in N_{G'_i}(I_{i+1})]$ is no less than a positive constant c_1.

Proof. Fix an arbitrary good vertex v in G'_i. By Lemma 6.28, the probability that $v \in N_{G'_i}(J_{i+1})$ is no less than a positive constant. In other words, the probability that some neighbor of v in G'_i belongs to J_{i+1} is no less than a positive constant. On the other hand, if y is a neighbor of v and y is in J_{i+1}, then the probability that $y \in I_{i+1}$ is no less than $\frac{1}{4}$ by step 2.1.2. Therefore, it seems that the probability that some neighbor of v in G'_i belongs to I_{i+1} is no less than a positive constant. However, this intuitive inference is not correct and we need a rigorous proof. Fortunately, by step 2.1.2, we have the fact that $\Pr[y \in I_{i+1} \mid y \in J_{i+1}] \geq \frac{1}{4}$ for every vertex y in G'_i. Using this fact, we can modify the proof of Lemma 6.28 given in [Koz92] to prove that $\Pr[v \in N_{G'_i}(I_{i+1})]$ is no less than a positive constant. Since the details are not so difficult, we omit them. ∎

We proceed to the proof of the fact that the expected value of t is $O(\log n)$. To show the fact, we use a potential function argument. Before describing our potential function, we observe that for every $0 \leq i \leq t$, each vertex y in G'_i satisfies that $N_G(y) \cap U_i$ is equal to $N_G(y) \cap X$. We denote the two vertices in $N_G(y) \cap X$ by x_1^y and x_2^y. Now, we are ready to define our potential function Ψ as follows: For every $0 \leq i \leq t$,

$$\Psi(G'_i, U_i) = \sum_{\text{edge } e = \{y,z\} \text{ in } G'_i} \psi(e, U_i),$$

where $\psi(e, U_i) = (2 - \mathrm{d}_{G[U_i]}(x_1^y))(2 - \mathrm{d}_{G[U_i]}(x_2^y))(2 - \mathrm{d}_{G[U_i]}(x_1^z))(2 - \mathrm{d}_{G[U_i]}(x_2^z))$.

Lemma 6.32 For every $0 \leq i \leq t-1$, $\mathcal{E}(\Psi(G'_i, U_i) - \Psi(G'_{i+1}, U_{i+1})) \geq c_2 \cdot \Psi(G'_i, U_i)$ for some positive constant c_2.

Proof. Fix an arbitrary i with $0 \leq i \leq t-1$. Let m'_i be the number of edges in G'_i. For each edge $e = \{y, z\}$ in G'_i, let $Z_e = \psi(e, U_{i+1})$ if e is also in G'_{i+1} and $Z_e = 0$ otherwise, and let B_e be the event that $\{y, z\} \cap N_{G'_i}(I_{i+1}) \neq \emptyset$. Then, we have

$$\Psi(G'_i, U_i) - \Psi(G'_{i+1}, U_{i+1}) = \sum_{\text{edges } e \text{ in } G'_i} (\psi(e, U_i) - Z_e).$$

Fix an arbitrary edge $e = \{y, z\}$ in G'_i. Note that $\psi(e, U_i) \geq 1$. Assume that event B_e occurs. Then, $y \in N_{G'_i}(I_{i+1})$ or $z \in N_{G'_i}(I_{i+1})$. By symmetry, we may assume that $y \in N_{G'_i}(I_{i+1})$. Let w be a vertex in I_{i+1} with $y \in N_{G'_i}(w)$. Then, by the construction of G',

either $\{y, w\}$ is an edge in G or $X \cap N_G(y) \cap N_G(w) \neq \emptyset$. In the former case, the edge $\{y, w\}$ is not contained in G'_{i+1} since it must be deleted from G' in step 2.3.3 during the $i+1$st iteration, and hence $\psi(e, U_i) - Z_e = \psi(e, U_i) \geq 1$. In the latter case, $\{x_1^y, x_2^y\} \cap \{x_1^w, x_2^w\} \neq \emptyset$ and we can simply show that $\psi(e, U_i) - Z_e = \psi(e, U_i) - \psi(e, U_{i+1}) \geq 1$. Thus, in both cases, we have $\psi(e, U_i) - Z_e \geq 1$. This implies that $\mathcal{E}(\psi(e, U_i) - Z_e \mid B_e) \geq 1$. Therefore, $\mathcal{E}(\psi(e, U_i) - Z_e) \geq \mathcal{E}(\psi(e, U_i) - Z_e \mid B_e)\Pr[B_e] \geq \Pr[B_e]$. If in addition e is good, then by Lemma 6.31, $\mathcal{E}(\psi(e, U_i) - Z_e) \geq \Pr[B_e] \geq c_1$ for some constant $c_1 > 0$. Combining this with the fact that G_{i+1} is a subgraph of G_i, we now have

$$\begin{aligned}
\mathcal{E}(\Psi(G'_i, U_i) - \Psi(G'_{i+1}, U_{i+1})) &= \mathcal{E}\Big(\sum_{\text{edges } e \text{ in } G'_i - G'_{i+1}} (\psi(e, U_i) - Z_e)\Big) \\
&= \sum_{\text{edges } e \text{ in } G'_i - G'_{i+1}} \mathcal{E}(\psi(e, U_i) - Z_e) \\
&\geq \sum_{\text{good edges } e \text{ in } G'_i - G'_{i+1}} \mathcal{E}(\psi(e, U_i) - Z_e) \\
&\geq \sum_{\text{good edges } e \text{ in } G'_i - G'_{i+1}} c_1 \geq \frac{m'_i}{2} c_1.
\end{aligned}$$

The last inequality follows from the fact that at least half the edges in any simple graph are good [Koz92, MR95]. On the other hand, we have $\Psi(G'_i, U_i) \leq 16m'_i$. Thus, $\mathcal{E}(\Psi(G'_i, U_i) - \Psi(G'_{i+1}, U_{i+1})) \geq \frac{c_1}{32}\Psi(G'_i, U_i)$. This completes the proof. ∎

Lemma 6.33 $\mathcal{E}t = O(\log n)$.

Proof. Note that $\Psi(G'_0, U_0) \leq 16\binom{n}{2}$ and that the while-loop is iterated until $\Psi(G'_i, U_i) < 1$. Thus, by Lemma 6.32 and Theorem 1.3 in [MR95], we immediately have that $\mathcal{E}t \leq \int_1^{16\binom{n}{2}} \frac{1}{c_2 x} dx = O(\log n)$. This completes the proof. ∎

Theorem 6.34 An MLFS in an n-vertex graph G can be found in $O(\log n)$ expected time with n^2 processors on a CRCW PRAM.

Proof. By Theorem 6.27, it suffices to prove that Algorithm *FOREST* runs in $O(\log n)$ expected time with n^2 processors. Since there are only $O(n)$ edges between X and Y in G, step 1 takes $O(\log n)$ time with n^2 processors. Thus, by Lemma 6.33, it remains to show that steps 2.1 through 2.3 can be implemented in $O(1)$ time with n^2 processors.

By Lemma 6.28, step 2.1.1 can be done in $O(1)$ time with n^2 processors. Steps 2.1.2 and 2.1.3 use no more resources than step 2.1.1. Thus, the implementation of step 2.1 takes $O(1)$ time with n^2 processors. Implementing step 2.2 in $O(1)$ time using n processors is trivial. Now, consider the implementation of step 2.3. Recall that during any iteration of the while-loop, each vertex y in G' satisfies that $N_G(y) \cap U = N_G(y) \cap X$. Hence, the algorithm may compute $N_G(y) \cap U$ before entering the while-loop, and this takes $O(\log n)$ time with n^2 processors. Therefore, step 2.3.1 can be done in $O(1)$ time with n processors. The implementation of step 2.3.2 is trivial. It is not difficult to implement step 2.3.3 in $O(1)$ time with n^2 processors. Step 2.3.4 uses no more resources than step 2.3.1. In total, the implementation of step 2.3 takes $O(1)$ time with n^2 processors. This completes the proof. ∎

6.3.4 The NC algorithm

In this section, we obtain an NC algorithm for the MLF problem by derandomizing the randomized algorithm in section 6.3.2. Recall that the randomized algorithm consumes random bits in steps 2.1.1 and 2.1.2. We now consider how many random bits are sufficient. It has been shown in [Lub86, Koz92] that step 2.1.1 needs only $2\log n + O(1)$ random bits. We claim that step 2.1.2 needs only 1 random bit. To see this, we replace steps 2.1.2 and 2.1.3 with the following steps:

2.1.2'. Partition J into two subsets J_1 and J_2 such that both $U \cup J_1$ and $U \cup J_2$ are LFS's in G. (*Comment:* This step is possible because each connected component of $G[U \cup J]$ is a cycle or path.)

2.1.3'. Randomly set I to be J_1 or J_2.

It is easy to see that even if we do the above replacement, the algorithm remains correct, and more importantly, Lemma 6.31 and Lemma 6.32 still hold. Step 2.1.3' needs only 1 random bit, and hence the modified algorithm uses only $2\log n + O(1)$ random bits. Therefore, we can derandomize it using standard techniques. Since the details are rather standard, we omit them. Summarizing up, we obtain the following theorem:

Theorem 6.35 An MLFS in an n-vertex graph G can be found in $O(\log^2 n)$ time with $O(n^4)$ processors on an EREW PRAM.

6.4 Remarks

Restricting to bipartite graphs or sparse graphs (such as planar graphs), the MLF problem can be easily reduced to the MPS problem and hence has more efficient NC algorithms than Algorithm *FOREST*. Since the details are rather easy, we omit them here.

In this section, we point out an interesting open question related to this work. We say that a bipartite graph $G = (X, Y, E)$ is *special* if every $y \in Y$ has exactly 3 neighbors (in X). Consider the following problem: Given a special bipartite graph $G = (X, Y, E)$, we wish to find a maximal subset S of Y such that $G[X \cup S]$ is acyclic. We call this problem the *maximal triangular cactus* (MTC) problem. The MTC problem can be viewed as an extension of the restricted MLF problem. We describe an interesting application of the MTC problem below.

The *maximum planar subgraph* problem is, given a graph $G = (V, E)$, to find a maximum subset F of E such that the graph (V, F) is planar. This problem has applications in circuit layout and graph drawing, but is unfortunately NP-hard. Thus, it is of interest to design approximation algorithms for this problem. Note that the maximum planar subgraph of an n-vertex graph contains at most $3n - 5$ edges and that a spanning tree has exactly $n - 1$ edges. From this fact, we obtain a trivial approximation algorithm which always computes a planar subgraph whose size is at least $\frac{1}{3}$ optimal. Only recently, Călinescu *et al.* [CFFK96] gave the first approximation algorithm better than the trivial one.

To describe Călinescu *et al.*'s approximation algorithm, we need several definitions. A *triangular cactus* is a graph whose cycles (if any) are triangles and such that all edges appear in some cycle. A *triangular cactus in a graph* G is a spanning subgraph of G which is a triangular cactus. A triangular cactus H in G is *maximal* if there is no other triangular cactus in G which is a supergraph of H. It is easy to see that a maximal triangular cactus in a given graph can be computed in polynomial time. We are now ready to describe Călinescu *et al.*'s approximation algorithm. Given a (connected) graph G, their algorithm computes a maximal triangular cactus H in G, and then computes a spanning tree T in the graph obtained from G by merging each

connected component of H into a single vertex. The output of the algorithm is the union of H and T. In [CFFK96], it is shown that the size of the planar subgraph output by this approximation algorithm is at least $\frac{7}{18}$ optimal. We wish to parallelize this algorithm. To reach our goal, we only need to answer the question whether a maximal triangular cactus in a given graph can be computed in NC.

We claim that the problem of computing a maximal triangular cactus in a given graph can simply be reduced to the MTC problem. The reduction is as follows. Given a graph $G = (V, E)$, we construct a *special* bipartite graph $G' = (X, Y, E')$, where $X = V$, Y is the set of all triangles in G, and E' consists of all $\{x, y\}$ such that the vertex x appears in the triangle y. Clearly, there is a one-to-one correspondence between maximal triangular cactuses in G and maximal subsets S of Y such that $G'[X \cup S]$ is acyclic. Thus, the claim holds. Therefore, Călinescu *et al.*'s approximation algorithm can be parallelized if the MTC problem can be solved in NC. However, we do not know whether the MTC problem is in NC or not. This is an open question.