1995

Parallel graph algorithms

Kyongsoo Jung
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Jung, Kyongsoo, "Parallel graph algorithms" (1995). *Master's Theses*. 1004.
DOI: <https://doi.org/10.31979/etd.nk3m-8u3y>
https://scholarworks.sjsu.edu/etd_theses/1004

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

PARALLEL GRAPH ALGORITHMS

A Thesis

Presented to

**The Faculty of the Department of
Mathematics and Computer Science**

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Jung, Kyongsoo

May, 1995

UMI Number: 1374595

Copyright 1995 by
Jung, Kyongsoo
All rights reserved.

UMI Microform 1374595
Copyright 1995, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI

300 North Zeeb Road
Ann Arbor, MI 48103

© 1995

Jung, Kyongsoo

ALL RIGHTS RESERVED

**APPROVED FOR THE DEPARTMENT OF
MATHEMATICS AND COMPUTER SCIENCE**

Dr. John Mitchem

Dr. Edward Schmeichel

Dr. Jeff Smith

APPROVED FOR THE UNIVERSITY

ABSTRACT
PARALLEL GRAPH ALGORITHMS

by Jung, Kyongsoo

This thesis presents two groups of parallel graph algorithms: one is the work-efficient algorithms and the other is the efficient algorithms in the class NC. Minimum Spanning Tree and Shortest Path algorithms are introduced as examples of work-efficient algorithms and $\Delta+1$ Coloring and Δ Coloring algorithms are considered as examples of efficient algorithms in the class NC.

For each given parallel algorithm, this paper shows its verification and analyzes its running time and the number of processors used. In addition, for each given problem, its parallel algorithm is compared with its sequential algorithm.

ACKNOWLEDGMENTS

I wish to express sincere thanks to Dr. John Mitchem for his helpful suggestions and patient guidance in the writing of this thesis and to my wife Hyewon for assisting me to concentrate my attention on this paper.

TABLE OF CONTENTS

Chapter		Page
1	Introduction	1
	1. 1 Definitions of Graph	1
	1. 2 Representation of Graphs	3
	1. 3 Parallelism	4
	1. 4 Definition of PRAM	5
2	Minimum Spanning Tree and Shortest Path	7
	2. 1 Definitions and notation	7
	2. 2 Sequential Algorithms	7
	2. 3 Parallel Algorithms	9
3	Components and All Pairs' Distances	22
	3. 1 Definitions and notation	22
	3. 2 Components Algorithm	23
	3. 3 All Pairs' Distances Algorithm	27

4	Parallel $(\Delta+1)$ -coloring of constant degree graphs	31
4.1	Definitions and notation	31
4.2	Sequential $(\Delta+1)$ coloring	31
4.3	Parallel $(\Delta+1)$ coloring	32
5	Parallel Δ -coloring algorithm	42
5.1	Introduction	42
5.2	Sequential Δ coloring	43
5.3	Parallel Δ coloring	48
	References	92

Chapter 1 Introduction

In this paper we study parallel algorithms for well-known graph problems, such as Minimum Spanning Tree, Shortest Path, $\Delta+1$ Coloring, and Δ Coloring. We are primarily interested in dividing sequential algorithms into two groups: those which are easily parallelizable and those which apparently are not parallelizable. Minimum Spanning Tree and Shortest Path algorithms are examples of easily parallelizable sequential algorithms. $(\Delta+1)$ Coloring and Δ Coloring algorithms are examples of sequential algorithms which apparently are not parallelizable or at least it is difficult to find a parallel version of them.

1.1 Definitions of Graph

A (undirected) graph G is a pair (V,E) , where V is a finite, non empty set, and E is a set of subsets of V of order two. The set V is called the vertex set of G , and its elements are called vertices. The set E is called the edge set of G , and its elements are called edges.

In a directed graph (or digraph) $G = (V,E)$, the edge set E consists of ordered pairs of vertices. An edge $(v,w) \in E$ is represented in diagrams as $v \rightarrow w$.

A subgraph of a graph or digraph $G=(V,E)$ is a graph or a digraph $G' = (V',E')$ such that $V' \subseteq V$ and $E' \subseteq E$. Given a set $V' \subseteq V$, the subgraph of G induced by V' , denoted by $G(V')$, is the graph $G' = (V',E')$, where $E' = \{\{v,w\} \in E : v,w \in V'\}$. A complete graph is a graph with an edge between each pair of vertices. Vertices v and w are said to be incident with the edge $\{v,w\}$ and vice versa.

If $\{v,w\}$ is an edge in a graph $G = (V,E)$, we say that vertex v is adjacent to vertex w . When the graph is directed, the adjacency relation is not necessarily symmetric. If however, the graph is undirected, the adjacency relation is symmetric. In a directed graph $G = (V,E)$, a neighbor of a vertex v is any vertex that is adjacent to v in the undirected version of G . That is, w is a neighbor of v if either $(v,w) \in E$ or $(w,v) \in E$. In an undirected graph, v and w are neighbors if they are adjacent.

The degree of a vertex v in a graph, denoted by $\deg(v)$, is the number of edges incident on it. In a directed graph, the out-degree of a vertex is the number of edges leaving it, and the in-degree of a vertex is the number of edges entering it. The degree of a vertex in a directed graph is its in-degree plus its out-degree.

A weighted graph (or weighted digraph) is a triple (V,E,W) , where (V,E) is a graph (or digraph) and W is a function from E into $\mathbb{Z}^+$, the positive integers. For an edge $\{v,w\}$, $W(\{v,w\})$ is called the weight of the edge $\{v,w\}$. Let $\{v,w\} \notin E(G)$. Then if v is equal to w , then $W(\{v,w\}) = 0$. Otherwise $W(\{v,w\}) = \infty$.

A path from v to w in a graph or digraph $G = (V,E)$ is a sequence of edges $\{v_0,v_1\}, \{v_1,v_2\}, \dots, \{v_{k-1},v_k\}$, such that $v_0 = v$, $v_k = w$, and $v_0,v_1,\dots,v_k$ are all distinct. The length of the path is k . A vertex v alone is considered to be a path of length zero from v to itself. If G is an weighted graph, the length of the path is the sum of the weights of the edges in the path. A cycle in a graph or digraph $G = (V,E)$ is like a path $v_0,v_1,\dots,v_k$, with $k \geq 2$, except that $v_0 = v_k$.

1.2 Representation of Graphs

Let $G = (V, E)$ be a graph or digraph with $V = \{v_1, v_2, \dots, v_n\}$ and $|E| = m$. Then G can be represented by a $n \times n$ matrix $A = (a_{ij})$, called the adjacency matrix for G . A is defined by

$$\text{for } 1 \leq i, j \leq n, a_{ij} = \begin{cases} 1, & \text{if } \{v_i, v_j\} \in E \\ 0, & \text{otherwise} \end{cases}$$

If $G = (V, E, W)$ is a weighted graph or digraph, the weights can be stored in the adjacency matrix by modifying its definition as follows:

$$\text{for } 1 \leq i, j \leq n, a_{ij} = \begin{cases} W(\{v_i, v_j\}), & \text{if } \{v_i, v_j\} \in E \\ 0, & \text{if } v_i = v_j \\ \infty, & \text{otherwise.} \end{cases}$$

An alternative to the adjacency matrix representation is a data structure called the adjacency list. An adjacency list contains, for each vertex v , a linked list indicating which vertices are adjacent to v . The data in the adjacency list will vary with the problem. The nodes in the linked lists have a vertex field and a link field, where vertex field contains a vertex number and link field contains a pointer. Each such node represents an edge in the graph or digraph. A pointer array for the n list heads, one for the adjacency list of each vertex, is also required. Additional fields may be added to the list heads or the linked list nodes as required by the algorithms to be used. For example, if a graph or digraph is weighted, a weight field is included in the list head and each node.

1.3 Parallelism

In this thesis, we consider parallel algorithms, algorithms that perform more than one operation at the same time in parallel, i.e., algorithms for machines that have more than one processor working on one problem at the same time. In order to study parallel algorithms, we must choose an appropriate model for parallel computing. Parallel machines have the potential to solve many problems much faster than sequential machines. However, many difficulties, both practical and theoretical, remain to be resolved. There are several general-purpose and special purpose formal models of parallel machines that correspond to various (real or theoretical) hardware designs. The parallel algorithms in this thesis are presented in terms of one popular theoretical model: the parallel random-access machine, or PRAM. Many parallel algorithms for arrays, lists, and graphs can be easily described in the PRAM model. Although the PRAM model ignores many important aspects of real parallel machines, it is now widely used and serves as a good tool for introducing parallel computing.

The running time of a parallel algorithm depends on the number of processors executing the algorithm as well as the size of the problem input. Therefore, we define the work performed by a parallel algorithm as the product of its running time and the number of processors it requires. We say that a parallel algorithm A is work-efficient with respect to sequential algorithm A' if the work done by A is an asymptotically tight bound for the running time of A' solving the same problem. We also say that a parallel algorithm is efficient if the algorithm takes polylogarithmic time using polynomially many processors. A polylogarithmic time algorithm takes $O(lg^k n)$ time for some constant integer k , where n

is the problem size. Problems which can be solved within these constraints are regarded as having efficient parallel solutions and said to belong to the class NC.

1.4 Definition of PRAM

A parallel random-access machine (PRAM) consists of p ordinary (serial) processors $P_1, P_2, \dots, P_p$, all of which have a shared, global memory for storage. All processors can read from or write to the global memory at the same time. The processors also have a private, local memory for their own computation. All the processors run the same program, but each processor knows its own index, and the program may instruct processors to do different things depending on their indices.

Each time step has three phases :

1. read phases, in which each processor P_i may read from a memory cell.
2. computation phases, in which each processor P_i may do some computation.
3. write phases, in which each processor P_i may write to a memory cell.

PRAM processors are synchronized. That is, they all begin each step at the same time, and all the processors that write at any step write at the same time.

A concurrent read algorithm is a PRAM algorithm during whose execution multiple processors can read from the same location of shared memory at the same time. An exclusive read algorithm is a PRAM algorithm in which no two processors ever read the same memory location at the same time. We make a similar distinction with respect to whether or not multiple processors can write into the same memory location at the same

time, dividing PRAM algorithms into concurrent write and exclusive write algorithms.

Therefore we divide the types of algorithms into four categories:

1. EREW : exclusive read and exclusive write,
2. CREW : concurrent read and exclusive write,
3. ERCW : exclusive read and concurrent write, and
4. CRCW : concurrent read and concurrent write.

The algorithms considered in this thesis are EREW or CRCW. EREW PRAM is a PRAM that supports only EREW algorithms, and one that supports CRCW algorithms is called a CRCW PRAM. A CRCW PRAM can execute any of the other 3 types of algorithms, but a EREW PRAM can execute only EREW algorithms.

When multiple processors write to the same location in a CRCW algorithm, the effect of the parallel write depends on one of the following four CRCW models:

1. common : all values written are the same,
2. arbitrary : an arbitrary value from among those written is actually stored,
3. priority : the value written by the lowest-indexed processor is stored, and
4. combining : the value stored is some specified combination of the values written.

All CRCW algorithms in this thesis use the priority-CRCW model, in which a low indexed processor has higher priority than a high indexed processor. Sometimes, however, processors will have more than one index. For example, assume that we have n^3 processors. Then each processor can be represented by P_{ijk} , for $1 \leq i, j, k \leq n$. We say that processor $P_{i'j'k'}$ has higher priority than processor $P_{ij'k'}$, if (i', j', k') is smaller lexicographically than (i, j, k) .

Chapter 2

Minimum Spanning Tree and Shortest Path

2.1 Definitions and notation

A spanning tree for a graph, $G=(V,E)$, is a subgraph of G that is a tree containing all the vertices of G . In a weighted graph the weight of a subgraph is the sum of the weights of the edges in the subgraph. A minimum spanning tree for a weighted graph is a spanning tree with minimum weight.

Given a weighted graph $G=(V,E,W)$, the shortest path problem is to find a "shortest" path between two specified vertices. A path from v to w is a shortest path from v to w if there is no path from v to w with lower weight. We define the distance from a vertex x to a vertex y , denoted by $d(x,y)$, to be the weight of a shortest path from x to y .

2.2 Sequential Algorithms

The minimum spanning tree algorithm and the shortest path algorithm, discovered by R.C.Prim(1957) and E.W.Dijkstra(1959), respectively, are very similar to each other in approach, and both of them are parallelizable sequential algorithms. Baase(1988) explains these algorithms in detail.

Prim's algorithm begins by selecting an arbitrary starting vertex, and then chooses a new vertex and edge at each iteration to branch out from the part of the tree constructed so far. For each iteration, a vertex belongs to exactly one of the following three categories :

Tree Vertex : in the tree constructed so far.

Fringe Vertex : not in the tree, but adjacent to some vertices in the tree.

Unseen Vertex : all others.

Prim's algorithm chooses an edge from a tree vertex to a fringe vertex of minimum weight, and selects the fringe vertex as a new tree vertex. The main steps of the algorithm can be described as follows.

Step 1: Select an arbitrary vertex to start the tree

Step 2: while there are fringe vertices do

 select an edge of minimum weight between a tree vertex and a
 fringe vertex

 add the selected edge and the fringe vertex to the tree

end

Dijkstra's algorithm accepts as an input two specific vertices, a starting vertex v and an end vertex w . Then using the starting vertex, the algorithm chooses a new vertex and edge at each iteration to branch out from the part of the tree constructed so far just like Prim's algorithm does. Only the difference between these algorithms is how they select a vertex from the fringe and one of its incident edges. For each Tree vertex x , Dijkstra's algorithm keeps track of the distance from the starting vertex to x , and uses the distance to select the next tree vertex. The main step of the algorithm is as follow:

when two input vertices are v and w ,

```
Step 1: while  $w$  is not tree vertex do
    select an edge  $\{x,y\}$  such that  $d(v,x)+W(\{x,y\}) =$ 
     $\min\{d(v,x)+W(\{x,y\}): x \text{ is a tree vertex and } y \text{ is a fringe vertex}\}$ 
    add the selected edge and fringe vertex to the tree
    assign  $d(v,x) + W(\{x,y\})$  to  $d(v,y)$ 
end
```

The key step in these algorithms is the selection of a vertex from the fringe and an incident edge, which can be parallelized. The following sections show parallel versions of these algorithms.

2.3 Parallel Algorithms

Here, we need to introduce a technique to find the smallest key (or the largest key) in the given list. In a tournament, elements are paired off and compared in rounds. In succeeding rounds, the winners from the previous round are paired off and compared. Therefore, the smallest key (or the largest key) is found in $\lceil \lg n \rceil$ rounds. All of the comparisons in one round can be performed at the same time. Thus the tournament method naturally gives us a parallel algorithm. Figure 2.1 shows an example of a tournament.

Figure 2.1. An example of a tournament. Each number in the circle is an element in the given list.

Let $G = (V, E, W)$ be a given weighted graph, where $V = \{1, 2, \dots, n\}$ and $W(e) \geq 0$ for each $e \in E$. And also let L_n denote $\lceil \lg n \rceil$. Then the next two algorithms shown in this section use L_n processors, say $P_1, \dots, P_{L_n}$, and run in $O(n^2 / L_n)$. Therefore, the work done by these algorithms is $L_n \times n^2 / L_n$ or n^2 , which is the worst case running time of sequential versions, and so we may say that these parallel algorithms are work-efficient with respect to the common sequential versions.

Since these algorithms use L_n processors, where the number of vertices is n , each processor P_i is in charge of the vertices in V_i , where V_i is a set of $\lceil n/L_n \rceil$ vertices. One possible way to get a set V_i is to let $V_i = \{j : (i-1) \times \lceil n/L_n \rceil < j \leq i \times \lceil n/L_n \rceil\}$. In this method, some dummy vertices can be introduced but those vertices do not affect the algorithms introduced in this section, since each dummy vertex is not a neighbor of any vertices in G .

During the runs of the parallel algorithms, a vertex may be thought of as being one of two types:

Tree Vertex : in the tree constructed so far

Fringe Vertex : all others

The parallel algorithms work just like the sequential versions except in the way a new tree vertex is selected. We note that the definition of fringe vertex here differs from that used on page 8.

First we consider the algorithm, Parallel-Minimum-Spanning-Tree. It starts from selecting an arbitrary vertex as a tree vertex. Then it iteratively chooses a new tree vertex, v , such that v was a fringe vertex and for a tree vertex u , $W(\{v, u\}) = \min\{W(\{x, y\}) : x \text{ is tree vertex, } y \text{ is fringe vertex}\}$. This greedy algorithm makes locally optimal choices at each step and produces a globally optimal solution. Figure 2.2 contains an example of the action of the algorithm, and Figure 2.3 shows the values of the global memories.

(a) The input graph G . Find a minimum spanning tree.

(b) After the first iteration, edge $\{1,4\}$ has been selected.

(c) After the fourth iteration, edge $\{5,6\}$ has been selected.

(d) After the last iteration, edge $\{3,8\}$ has been selected.

Figure 2.2. An example for the minimum spanning tree algorithm. The solid vertices are tree vertices. All vertices have been prefixed by a character "v" in distinction from the weight of each edge.

Processor	P ₁			P ₂			P ₃		
Vertex	1	2	3	4	5	6	7	8	9
dist	∞	6	∞	4	∞	9	∞	∞	∞
parent	1	1	-	1	-	1	-	-	-
status	tree	fringe	fringe	tree	fringe	fringe	fringe	fringe	fringe
Candidate	2			4			7		
M	6			4			∞		
NewVx	4								

(a) after the first iteration

Processor	P ₁			P ₂			P ₃		
Vertex	1	2	3	4	5	6	7	8	9
dist	∞	∞	3	∞	∞	1	2	∞	∞
parent	1	4	5	1	4	5	5	-	-
status	tree	tree	fringe	tree	tree	tree	fringe	fringe	fringe
Candidate	3			6			7		
M	3			1			2		
NewV _x	6								

(b) after the forth iteration

Processor	P ₁			P ₂			P ₃		
Vertex	1	2	3	4	5	6	7	8	9
dist	∞	∞	∞	∞	∞	∞	∞	5	∞
parent	1	4	5	1	4	5	5	3	-
status	tree	tree	tree	tree	tree	tree	tree	tree	fringe
Candidate	1			4			8		
M	∞			∞			5		
NewVx	8								

(c) after the last iteration

Figure 2.3. Variation of a value of the global memories for the example in Figure 2.2.

Algorithm Parallel-Minimum-Spanning-Tree (G)

{ Input : a weighted graph $G = (V, E, W)$, where $V = \{1, 2, \dots, n\}$

Output : minimum spanning tree $T = (V', E')$

The number of processors used is $\lceil \lg n \rceil$.

Global variables used :

NewVx : a vertex which was added into the tree T most recently.

Parent : an array of size n which is used to add an edge into the tree T.

M : an array of size $\lceil \lg n \rceil$ which is used to compute the minimum value among the $\lceil \lg n \rceil$ candidates coming from each processor.

}

1: $V' = \emptyset, E' = \emptyset$

{ initialization }

2: for each processor P_i , in parallel

3: for each vertex x in V_i do { V_i as defined on the previous page }

4: $\text{dist}[x] = \infty$

 { add a vertex 1 into the tree T }

5: for each processor P_i in parallel

6: if i is equal to 1

7: $V' = V' \cup \{ 1 \}$

8: NewVx = 1

 { main loop }

9: for $p = 1$ to $|V|$ do

10: for each processor P_i in parallel

11: for each x in V_i do

12: if (x is in tree)

```

13:          dist[x] = ∞    { if x is in tree, then make x lose on every
                           comparison by assigning a huge number to x }
14:          else if (dist[x] > W({NewVx,x}))
               { update distance from NewVx to x }
15:          dist[x] = W({NewVx,x})
16:          parent[x] = NewVx
17:          let Candidate be an index such that dist[Candidate] =
               min{dist[x] : x ∈ Vi}.
18:          M[i] = dist[Candidate]
               { Now M[i] is the minimum value among distances between tree vertices and
                 fringe vertices in Vi }
19:          Let Min be min{M[i] : 1 ≤ i ≤ ⌈lg n⌉}
20:          for each processor Pi, in parallel
21:              if dist[Candidate] = Min
22:                  NewVx = Candidate          { may require CRCW }
23:          for each processor Pi, in parallel
24:              if ( NewVx ∈ Vi )
25:                  V' = V' ∪ {NewVx}          { construct the tree T }
26:                  E' = E' ∪ { {NewVx, parent[NewVx] } }

```

Most of the steps in this algorithm run in constant time except the "for" loops, line 3, 9, and 11, and the statements to find a minimum value, line 17 and 19. However, line 17 and line 19 calculate a minimum value among $n/\lceil \lg n \rceil$ values and $\lceil \lg n \rceil$ values, respectively. Since line 17 runs sequentially on $n/\lceil \lg n \rceil$ elements, it takes $O(n/\lceil \lg n \rceil)$ time. On the other hand, since line 19 runs in parallel on $\lceil \lg n \rceil$ elements, it takes

$O(\lceil \lg(\lg n) \rceil)$ time. The "for" loops, line 3, 9, and 11, iterate $n/\lceil \lg n \rceil$ times, $(n-1)$ times, and $n/\lceil \lg n \rceil$ times, respectively. Therefore, the running time of the algorithm Parallel-Minimum-Spanning-Tree is in $O(n^2/\lceil \lg n \rceil)$. Since this algorithm uses $\lceil \lg n \rceil$ processors and runs in $O(n^2/L_n)$, the work done by this algorithms is $O(n^2)$, which is the worst case running time of sequential versions. Therefore, we may say that the algorithm Parallel-Minimum-Spanning-Tree is work-efficient with respect to the usual implementation. The required PRAM model is obviously a CRCW PRAM due to line 22.

Now we consider the second algorithm, Parallel-Shortest-Path. It accepts as an input two specific vertices v and w , where we use v as a starting vertex and w as an end vertex, and tries to find a shortest vw path. For each iteration, only one fringe vertex u becomes a new tree vertex, and u is selected because $d(v,u) \leq d(v,i)$ for all fringe vertex i . To implement this idea, Parallel-Shortest-Path algorithm keeps track, for each vertex x , its currently known distance from v , denoted by $\text{dist}[x]$, and selects one vertex among the fringe vertices, having the least distance from v . For example, let's assume that $d(v,k) = 7$, where k just became a tree vertex. Also assume that vertices i, j are the only fringe vertices, and $W(\{k,i\}) = 2$, current $\text{dist}[i] = 10$, $W(\{k,j\}) = \infty$, and current $\text{dist}[j] = 8$. Then since current $\text{dist}[i]$ is greater than $d(v,k) + W(\{k,i\})$, the algorithm assigns $d(v,k) + W(\{k,i\})$ to $\text{dist}[i]$ so that new $\text{dist}[i] = 9$. On the other hand, the value of $\text{dist}[j]$ is not changed because $\text{dist}[j]$ is less than $d(v,k) + W(\{k,j\})$. Then the algorithm chooses vertex j as a new tree vertex since $\text{dist}[j] < \text{dist}[i]$.

(a) The input graph G . Find a shortest path from vertex 1 to vertex 7.

(b) After the first iteration, edge $\{1,4\}$ has been selected. ($d(1,4) = 4$)

(c) After the third iteration, edge $\{4,5\}$ has been selected. ($d(1,5) = 7$)

(d) After the last iteration, edge $\{5,7\}$ has been selected. ($d(1,7) = 9$)

Figure 2.4. An example for the shortest path algorithm. The solid vertices are tree vertices. All vertices have been prefixed by a character "v" in distinction from the weight of each edge.

Processor	P ₁			P ₂			P ₃		
Vertex	1	2	3	4	5	6	7	8	9
dist	∞	6	∞	4	∞	9	∞	∞	∞
d(v,x)	0	-	-	4	-	-	-	-	-
parent	1	1	-	1	-	1	-	-	-
status	tree	fringe	fringe	tree	fringe	fringe	fringe	fringe	fringe
Candidate	2			4			7		
M	6			4			∞		
NewVx	4								

(a) after the first iteration

Processor	P ₁			P ₂			P ₃		
Vertex	1	2	3	4	5	6	7	8	9
dist	∞	∞	12	∞	7	9	∞	∞	∞
d(v,x)	0	6	-	4	7	-	-	-	-
parent	1	1	2	1	4	1	-	-	-
status	tree	tree	fringe	tree	tree	fringe	fringe	fringe	fringe
Candidate	3			5			7		
M	12			7			∞		
NewVx	5								

(b) after the third iteration

Processor	P ₁			P ₂			P ₃		
Vertex	1	2	3	4	5	6	7	8	9
dist	∞	∞	10	∞	∞	∞	9	∞	∞
d(v,x)	0	6	-	4	7	8	9	-	-
parent	1	1	5	1	4	5	5	-	-
status	tree	tree	fringe	tree	tree	tree	tree	fringe	fringe
Candidate	3			4			7		
M	10			∞			9		
NewVx	7								

(c) after the last iteration

Figure 2.5. Variation of a value of the global memories for the example in Figure 2.4.

To trace the shortest path, we use an array called parent as follows: If $\text{dist}[x]$ is greater than $d(v,k) + W(\{k,x\})$, then $\text{dist}[x]$ is changed to $d(v,k) + W(\{k,x\})$ and k is assigned to $\text{parent}[x]$. After the algorithm terminates, we can get the shortest path by traversing the array parent from destination, vertex w , back to the initial vertex v .

Figure 2.4 shows the process of some iterations for the given input graph, and Figure 2.5 shows the values in the global memories corresponding to Figure 2.4.

Algorithm Parallel-Shortest-Path(G, v, w)

{ Input : a weighted graph $G = (V, E, W)$, where $V = \{1, 2, \dots, n\}$, and two specific vertices v and w .

Output : shortest vw path , a subgraph of $T = (V', E')$ which is a digraph constructed by this algorithm

The number of processors used is $\lceil \lg n \rceil$.

Global variables used :

NewVx : a vertex which has added into the tree T most lately.

Parent : an array of size n which is used to add an edge into the tree T .

M : an array of size $\lceil \lg n \rceil$ which is used to compute the minimum value among the $\lceil \lg n \rceil$ candidates coming from each processor.

}

1: $V' = \emptyset, E' = \emptyset$

{ initialization }

2: for each processor P_i , in parallel

3: for each vertex x in V_i do

4: $\text{dist}[x] = \infty$

```

    { add an the initial vertex v into the tree T }
5:   for each processor  $P_i$  in parallel
6:       if  $V_i$  contains v
7:            $V' = V' \cup \{ v \}$ 
8:            $NewV_x = v$ 
9:            $d(v,v) = 0$ 
    { main loop }
10:  while w is a fringe vertex
11:      for each processor  $P_i$  in parallel
12:          for each x in  $V_i$  do
13:              if (x is in tree)
14:                   $dist[x] = \infty$     { if x is in tree, then x never yields the minimum
                                         in line 18 or 20 }
15:                  else if ( $dist[x] > W(\{NewV_x, x\}) + d(v, NewV_x)$ )
                                         { update distance from  $NewV_x$  to x }
16:                       $dist[x] = W(\{NewV_x, x\}) + d(v, NewV_x)$ 
17:                       $parent[x] = NewV_x$ 
18:                  let Candidate be the index such that  $dist[Candidate] =$ 
                      $\min\{dist[x] : x \in V_i\}$ .
19:                   $M[i] = dist[Candidate]$ 
    { Now  $M[i]$  is the minimum value among distances between tree vertices and
      fringe vertices in  $V_i$  }
20:  Let Min be  $\min\{M[i] : 1 \leq i \leq \lceil \log n \rceil\}$ 
21:  for each processor  $P_i$ , in parallel
22:      if  $dist[Candidate] = Min$ 

```

```

23:          NewVx = Candidate          { may require CRCW }
24:    for each processor  $P_i$ , in parallel
25:      if ( NewVx  $\in V_i$  )
26:         $V' = V' \cup \{\text{NewVx}\}$           { construct the tree T }
27:         $E' = E' \cup \{\{\text{NewVx}, \text{parent}[\text{NewVx}]\}\}$ 
28:         $d(v, \text{NewVx}) = \text{dist}[\text{NewVx}]$ 
        { Output the path. The vertices will be listed in reverse order, i.e., from w to v. }
29:  for each processor  $P_i$ , in parallel
30:    if (  $w \in V_i$  )
31:      output (w)
32:      temp = w
33:      while temp  $\neq v$ 
34:        temp = parent[temp]
35:        output (temp)
36:      output (temp)

```

The running time of this algorithm is equal to that of Parallel-Minimum-Spanning-Tree algorithm, $O(n^2/\lceil \log n \rceil)$, and so it is work-efficient with respect to the usual implementation to Dijkstra's algorithm. The required PRAM model is also a CRCW PRAM.

Chapter 3

Components and All Pairs' Distances

3.1 Definitions and notation

An undirected graph $G=(V,E)$ is said to be connected if there exists a path between every pair of distinct vertices in G . A component of a graph G is a subgraph G' of G such that G' is connected and not contained in any other connected subgraph H of G . In this chapter, we consider parallel algorithms to find components of an undirected graph G , and to find the shortest paths between every pair of vertices of a positive weighted graph G' , where the graph G and G' are given by their adjacency matrix.

For a matrix M , M_{ij} denotes the element of M in the i^{th} row and j^{th} column. We say that a matrix C represents all paths in G of length k or less if and only if

$$C_{ij} = 1, \text{ if vertex } i \text{ and vertex } j \text{ are joined by a path of length at most } k, \text{ and}$$
$$C_{ij} = 0, \text{ otherwise.}$$

Thus replacing the 0's of the main diagonal of the adjacency matrix with 1's results in the matrix C which represents all paths of G of length 1 or less. If a matrix C represents all paths in G of length $n-1$ or less, where $n = |V(G)|$, then C is called a connectivity matrix of G .

3.2 Components Algorithm

Given an adjacency matrix A of a graph $G=(V,E)$, we first compute a connectivity matrix C . Then we use the matrix C to find the components. These algorithms use n^3 processors on a CRCW PRAM and terminate in $O(\log n)$ time, where n is the size of V . We are only interested in whether or not there exists a path between two vertices. Therefore, throughout this section we use only Boolean operations.

The first algorithm, Connectivity, accepts as an input the adjacency matrix A of a graph G and computes the connectivity matrix C . This algorithm is based on the following lemma.

Lemma 3.1. Let B be the $n \times n$ matrix representing all paths in G of length 1 or less. Then B^k , produced by Boolean matrix multiplication, represents paths in G of length k or less for each k , $1 \leq k$.

Proof. We show this lemma by induction on k . By the definition of matrix B , the result is immediate when $k = 1$. Assume that for some m , B^m represents paths in G of length m or less. Now we want to show that B^{m+1} represents paths in G of length $m+1$ or less. Since $B^{m+1} = B^m \times B$, $B^{m+1}_{ij} = \sum B^m_{ik} \times B_{kj}$, for each entry (i,j) of a matrix B^{m+1} . If there is a path of length $m+1$ or less between vertices i and j , there exists a path of length m or less from i to one of neighbors of j , say l . Then both B^m_{il} and B_{lj} have value 1, and thus so does B^{m+1}_{ij} . If there is no path of length $m+1$ or less between i and j , then there must not exist a path of length m or less from i to any neighbors of j . Therefore, in this case we may say $B^m_{ik} = 0$, if vertex k is adjacent to vertex j , and

$B_{kj} = 0$, otherwise. Thus B^{m+1}_{ij} has value 0. Therefore the result is true for $m+1$ and by induction, the lemma is true. ■

The lemma implies that we can easily get the connectivity matrix C by the simple following two steps :

1. finding the matrix B representing all paths in G of length less than 2, and
2. computing $C = B^k$, where k is the smallest power of two which is greater than or equal to $n-1$.

Algorithm Connectivity (A)

{ Input : an adjacency matrix A of a graph $G = (V,E)$.

Output : a connectivity matrix C .

Uses n^3 processors and runs in $O(\lg n)$ time.

}

{ Find the matrix B representing all paths in G of length 1 or less }

1: for each processor P_{ij1} in parallel

2: $B_{ij} = A_{ij}$

3: for each processor P_{i11} in parallel

4: $B_{ii} = 1$

{ Compute connectivity matrix C }

5: for $l = 1$ to $\lceil \log(n-1) \rceil$ do

6: for each processor P_{ij1} in parallel

7: $C_{ij} = 0$

```

      { Compute Boolean matrix multiplication }
8:      for each processor  $P_{ijk}$  in parallel
9:          if  $(B_{ik}B_{kj} = 1)$   $C_{ij} = 1$ 
10:      for each processor  $P_{ijl}$  in parallel
11:           $B_{ij} = C_{ij}$ 

```

Theorem 3.1. The algorithm Connectivity runs in $O(\log n)$, using n^3 processors on a CRCW PRAM, where n is $|V|$ and V is the set of vertices of a given graph G .

Proof. The running time of this algorithm is $O(\log n)$ since the "for" loop in line 5 iterates $\lceil \log(n-1) \rceil$ times and the other statements are done in constant time. Boolean matrix multiplication in line 8 ~ 9 uses n^3 processors and requires a concurrent read priority write PRAM. ■

The next algorithm, Components, accepts as an input an adjacency matrix A of a graph G , runs the algorithm Connectivity, and then uses the output, matrix C , of the algorithm Connectivity to output an n vector D such that $D(i) = D(j)$ if and only if vertex i is in the same component as vertex j . Furthermore $D(i)$ is one of the vertices of the component containing i . For example, if $V = \{1,2,3,4,5\}$ and $E = \{\{1,3\}, \{3,4\}, \{2,5\}\}$, then $D = (1,2,1,1,2)$. After finding the connectivity matrix C , it is easy to find the vector D . All vertices in the same component have the same row vectors in matrix C . For instance, assume that a component consists of three vertices, i , j , and k , where $i < j < k$. Then only C_{vi} , C_{vj} , and C_{vk} have value 1 in row v , where $v = i, j$, or k . Therefore, by letting $D(v) = i$, for $v = i, j$, or k , we can begin to build the output vector D .

Algorithm Components (A)

{ Input : an adjacency matrix A of a graph $G = (V, E)$.

Output : an n vector D such that $D(i) = D(j)$ if and only if vertex i is in the same component as vertex j .

Uses n^3 processors and runs in $O(\lg n)$ time.

}

{ Compute the connectivity matrix }

1: Run the algorithm Connectivity(A)

{ Construct the vector D }

2: for each processor P_{i11} in parallel

3: $D(i) = 0$

4: for each processor P_{ij1} in parallel

5: if ($C_{ij} = 1$) $D(i) = j$

Theorem 3.2. The algorithm Components computes the components of an input graph G in $O(\log n)$, using n^3 processors on a CRCW PRAM.

Proof. The running time of this algorithm is dominated by the running time of the algorithm Connectivity which uses n^3 processors in time $O(\log n)$. As stated earlier, our concurrent write uses a priority write: When P_{ij} and P_{ik} , $j < k$, both write into memory $D(i)$, P_{ij} has priority. ■

3.3 All Pairs' Distances Algorithm

The algorithm All-Pairs'-Distances takes as input an $n \times n$ matrix M which is an adjacency matrix of a weighted graph $G = (V, E, W)$, and outputs the matrix M' defined as follows:

for $1 \leq i, j \leq n$

$M'_{ij} = 0$, if $i = j$

M'_{ij} = the sum of the weights of the edges in a shortest path between vertex i and vertex j , if $i \neq j$.

We may find M'_{ij} , for $i \neq j$, by calculating the minimum of sums of all possible sequences of edges between i and j . For each iteration, the search for a shorter path between each pair of vertices is extended to consider all possible paths utilizing up to twice the number of currently considered edges. Thus a logarithmic number of iterations is sufficient. We show how the algorithm, All-Pairs'-Distances, works by explaining a simple example. Assume that we have an weighted input graph $G = (V, E, W)$ represented by the adjacency matrix M , described in Figure 3.1. We denote $M^{(t)}$ as a modified matrix of M after t iterations. Since $M^{(t+1)}_{ij} = \min\{M^{(t)}_{ij}, M^{(t)}_{i1} + M^{(t)}_{1j}, M^{(t)}_{i2} + M^{(t)}_{2j}, \dots, M^{(t)}_{in} + M^{(t)}_{nj}\}$, we can see $M^{(1)}_{14} = \min\{\infty, \infty, 7, 10, \infty\} = 7$, and $M^{(2)}_{14} = \min\{7, 7, 4, 4, 7\} = 4$. The matrices $M^{(1)}$ and $M^{(2)}$ are also given in Figure 3.1.

(a) Input weighted graph G.

0	2	9	∞
2	0	1	5
9	1	0	1
∞	5	1	0

(b) Given adjacency matrix M.

0	2	3	7
2	0	1	2
3	1	0	1
7	2	1	0

(c) Matrix M, after 1st iteration.

0	2	3	4
2	0	1	2
3	1	0	1
4	2	1	0

(b) Matrix M, after 2nd iteration.

Figure 3.1. An example for the All-Pairs'-Distances algorithm.

Algorithm All-Pairs'-Distance (M)

{ Input : an $n \times n$ matrix M which is an adjacency matrix of a weighted graph G.

Output : a matrix M' defined as follows:

for $1 \leq i, j \leq n$

$M'_{ij} = 0$, if $i = j$

M'_{ij} = the sum of the weights of the edges in a shortest path
between vertex i and vertex j, if $i \neq j$.

Uses n^4 processors and runs in $O(\lg n)$ time.

}

```

1:   for each processor  $P_{ij11}$  in parallel
2:        $M'_{ij} = M_{ij}$ 
3:   for  $t = 1$  to  $\lceil \log n \rceil$  do
4:       for each processor  $P_{ijk1}$  in parallel
5:            $q_{ijk} = M'_{ij} + M'_{jk}$ 
            $\{M^{(t+1)}_{ij} = \min\{M^{(t)}_{ij}, M^{(t)}_{i1} + M^{(t)}_{1j}, M^{(t)}_{i2} + M^{(t)}_{2j}, \dots, M^{(t)}_{in} + M^{(t)}_{nj}\}.$ 
           Uses  $n^4$  processors to find each  $M^{(t+1)}_{ij}$  since finding a minimum number
           on  $n$  elements takes constant time, using  $n^2$  processors.(refer to pp.704-
           705 in Cormen,Leiserson,and Rivest(1990))
6:       for each processor  $P_{ijkl}$  in parallel
7:            $M'_{ij} = \min\{M'_{ij}, q_{i1j}, q_{i2j}, \dots, q_{inj}\}$ 
8:   for each processor  $P_{ij11}$  in parallel
9:       if  $i \neq j$ 
10:           $M_{ij} = M'_{ij}$ 
11:       else  $M_{ij} = 0$ 

```

Theorem 3.3. For $t = 1, 2, \dots, \lceil \log n \rceil$, $M^{(t)}_{ij}$ is the length of a shortest ij path with at most 2^t edges in the given graph G .

Proof. After the first iteration ($t = 1$), $M^{(1)}_{ij}$ is $\min\{M_{ij}, M_{i1} + M_{1j}, M_{i2} + M_{2j}, \dots, M_{in} + M_{nj}\}$, for $1 \leq i, j \leq n$. Therefore, $M^{(1)}_{ij}$ obviously is the length of a shortest ij path containing at most two edges. Assume that after t iteration, where $t < \lceil \log n \rceil$, $M^{(t)}_{ij}$ is the length of a shortest ij path containing at most 2^t edges. Let a sequence S of vertices, $v_0, v_1, v_2, \dots, v_k$, be a shortest path with at most 2^{t+1} edges between vertices i and j , where $v_0 = i$ and $v_k = j$.

If $k \leq 2^t$, then $M^{(t)}_{ij}$ is the length of S by inductive assumption. We need to show that $M^{(t+1)}_{ij} = M^{(t)}_{ij}$. We note that $M^{(t+1)}_{ij} = \min\{M^{(t)}_{ij}, M^{(t)}_{i1} + M^{(t)}_{1j}, M^{(t)}_{i2} + M^{(t)}_{2j}, \dots, M^{(t)}_{in} + M^{(t)}_{nj}\}$. Assume that $M^{(t)}_{ij} > M^{(t)}_{ip} + M^{(t)}_{pj}$, for some p , $1 \leq p \leq n$. Then $M^{(t+1)}_{ij} = M^{(t)}_{ip} + M^{(t)}_{pj}$, i.e., there is a ij path via p whose length is at most $M^{(t)}_{ip} + M^{(t)}_{pj}$. Since by the assumption $M^{(t)}_{ip}$ and $M^{(t)}_{pj}$ are the lengths of the shortest ip path and the shortest pj path, respectively, containing at most 2^t edges, the given sequence S can not be a shortest ij path with at most 2^{t+1} edges, which is contradiction by inductive assumption.

If $2^t < k \leq 2^{t+1}$, there exists the $(2^t+1)^{\text{th}}$ vertex v_{2^t} , say l , in the sequence S . We divide the sequence S into two sub sequences: $S_1 = v_0, v_1, \dots, v_{2^t}$ and $S_2 = v_{2^t}, v_{2^t+1}, \dots, v_k$. Since S is a shortest ij path, S_1 is a shortest il path with at most 2^t edges and S_2 is also a shortest lj path with at most 2^t edges. By the assumption $M^{(t)}_{il}$ and $M^{(t)}_{lj}$ are the lengths of the shortest il path and the shortest lj path. Therefore, we only need to show that $M^{(t+1)}_{ij} = M^{(t)}_{il} + M^{(t)}_{lj}$. Assume that $M^{(t+1)}_{ij} < M^{(t)}_{il} + M^{(t)}_{lj}$.

Case 1: $M^{(t+1)}_{ij} = M^{(t)}_{ij}$. Then there is a ij path using at most 2^t edges of length less than the length of S , which is contradiction.

Case 2: $M^{(t+1)}_{ij} = M^{(t)}_{ip} + M^{(t)}_{pj}$, for $p \neq l$. Then there is a ij path of length less than the length of S , which is also contradiction.

Therefore $M^{(t+1)}_{ij} = M^{(t)}_{il} + M^{(t)}_{lj}$. ■

Corollary 3.1. Let $s = \lceil \log n \rceil$. Then $M^{(s)}_{ij}$ is the length of a shortest ij path in the given graph G .

Proof. The desired conclusion comes from Theorem 3.3. ■

Chapter 4

Parallel $(\Delta+1)$ -Coloring of Constant Degree Graphs

4.1 Definitions and notation

In this section, the Vertex Covering problem and Maximal Independent set are defined, and the notation used in this chapter is introduced.

Given graph G , a coloring of the graph is an assignment of a color to each vertex such that no two neighboring vertices have the same color. The vertex coloring problem is to find a valid coloring that uses at most $\Delta+1$ colors. An independent set is a set of vertices $I \subseteq V(G)$ such that no two vertices in I are adjacent. The maximal independent set problem is to find an independent set I such that if any vertex v in $V(G)-I$ is inserted into I , then the set I is no longer independent.

Here, we assume that the given graph $G = (V,E)$, where the maximum degree is a constant Δ , and $V = \{0,1,2,\dots,n-1\}$, is represented by an adjacency list.

4.2 Sequential $(\Delta+1)$ Coloring

Let C be the set of colors of cardinality $\Delta+1$. Then for each vertex i in V , there are at most Δ vertices adjacent to i , i.e., at least one color can be assigned to i . Therefore, the sequential algorithm to color a graph with $\Delta+1$ colors is as simple as the following :

```

for i = 0 to n-1
    C' = C
    while there is a colored vertex j adjacent to i
        C' = C' - {color of j}
    color i with a color c ∈ C'
end

```

Since G has a constant maximum degree, the running time of the above algorithm is $O(n)$.

4.3 Parallel $(\Delta+1)$ Coloring

In this section, three algorithms from Goldberg and Plotkin (1987) are presented. Given a graph G with a constant maximum degree Δ , the first algorithm, Color-Constant-Degree-Graph, colors G with a constant number of colors. The second algorithm, Max-Independent-Set, uses this coloring to find a maximal independent set in the graph. The third algorithm, $(\Delta+1)$ -Coloring, iteratively applies the second one to find a maximal independent set in the graph and recolors the graph using only $\Delta+1$ colors.

$(\Delta+1)$ -Coloring algorithm uses an EREW PRAM of n processors, where each processor is capable of executing simple word and bit operations. For this algorithm, it is useful to assume that the word width is $\lceil \lg n \rceil$. The word operations include bit-wise Boolean operations, integer comparisons, and unary-to-binary conversion.

We denote that $\lg x = \log_2 x$, $\lg^{(1)} x = \lg x$, $\lg^{(i)} x = \lg \lg^{(i-1)} x$, and $\lg^* x = \min \{i \mid \lg^{(i)} x \leq 2\}$ and show that the running time of $(\Delta+1)$ -Coloring algorithm is $O(\lg^* n)$.

The first algorithm, Color-Constant-Degree-Graph, accepts as input a graph $G=(V,E)$ with constant maximum degree Δ , and uses n processors, $P_0, \dots, P_{n-1}$. Let $V(G) = \{0, 1, 2, \dots, n-1\}$, where each integer is written in binary with sufficient zeros on the left so that each integer is $L = \lceil \lg n \rceil$ bits long, and also let processor P_i be responsible for vertex i . The bits for each color will be indexed from left to right $0, 1, 2, \dots$. Initially color each vertex v with C_v which is the binary number representing vertex v . This algorithm iteratively recolors the graph G , each time reducing the upper bound on the number of colors.

In order to describe the recoloring, we consider a small example. Assume that $L = 5$, $\Delta = 3$, and vertex v colored 01101 is adjacent only to u colored 01000 and w colored 01100 . Then the index of the first bit in which C_v and C_u differ is 2 , and the value of the v 's bit at this position, denoted by $C_v(2)$, is 1 . In order to represent the various indices in binary, we need 3 bits because $\lceil \lg L \rceil = 3$. We, therefore, represent the difference between v and u by $\langle 010, 1 \rangle$, where $010 = (2)_2$ is the index of the first bit where u and v differ. Similarly, the difference between v and w is represented by $\langle 100, 1 \rangle$ because bit 4 is the first bit where v and w differ and v 's forth bit, $C_v(4)$, is 1 . Since Δ is 3 and degree of v is 2 , we need one more pair for consistency. So insert $\langle 000, 0 \rangle$. Then we have the list 010110010000 as the new color for vertex v . In this particular example the number of bits in the representation of the color of v has increased. However, in the actual algorithm the number of bits in the representation of the color for each vertex decreases at each iteration because the algorithm halts when this decrease is no longer generated. As we can see, the size of the list for each vertex is $\Delta \lceil \lg L + 1 \rceil$. If $L \leq \Delta \lceil \lg L + 1 \rceil$, then this algorithm terminates because otherwise the number of bits in the color used no longer

necessarily decreases. For the above example, therefore, the algorithm would terminate with the color of v , 01101, rather than recolor v with 011110110000.

In summary, at each iteration the algorithm recolors vertex v with a binary number with fewer bits. The new color of each vertex is a list of differences between the vertex and its neighbors. Each element of the list is an ordered pair that consists of the index of a position in which the color of the vertex differs from the color of one of its neighbors and the value of the bit at this position. This algorithm halts when the upper bound on the number of new colors is not less than that on the number of currently used colors, i.e., $L \leq \Delta(\lceil \lg L \rceil + 1)$.

To show that the running time of $(\Delta+1)$ -Coloring (G) is in $O(\lg^* n)$ time, we first analyze the algorithm Color-Constant-Degree-Graph.

Algorithm Color-Constant-Degree-Graph (G, L)

{ Input : a graph $G=(V, E)$ with constant maximum degree Δ , where $V = \{0, 1, 2, \dots, n-1\}$.

Output : a coloring of G with a constant number of colors, and L which is a size of bits expressing the number of colors.

Uses n processors, $P_0, \dots, P_{n-1}$.

C_i : a bit representation of color of the vertex i .

N_i : a set of neighbors of the vertex i .

}

1: $L = \lceil \lg n \rceil \quad \{ n = |V(G)| \}$

```

2:   for each processor  $P_i$  in parallel do
      { binary representation of  $i$  with  $L$  bits }
3:    $C_i \leftarrow \text{bin}(i, L)$       { initial coloring }
4:   while  $L > \Delta \lceil \lg L + 1 \rceil$  for each processor  $P_i$ , in parallel
5:      $N_i \leftarrow \{ j : (i, j) \in E \}$ 
6:     for  $k \leftarrow 1$  to  $|N_i|$  do
7:        $j \leftarrow N_i(k)$       {  $k^{\text{th}}$  element in  $N_i$  }
          { find the index of the first bit at which  $C_i$  and  $C_j$  differ }
8:        $\text{index} \leftarrow \min\{\text{index}' : C_i(\text{index}') \neq C_j(\text{index}')\}$ 
          { binary representation of index with  $\lceil \lg L \rceil$  bits }
9:        $i_k \leftarrow \text{bin}(\text{index}, \lceil \lg L \rceil)$ 
10:       $b_k \leftarrow C_i(i_k)$ 
11:      for  $k \leftarrow |N_i| + 1$  to  $\Delta$  do { add dummy bits for consistency. }
12:        { binary representation of index with  $\lceil \lg L \rceil$  bits }
           $i_k \leftarrow \text{bin}(0, \lceil \lg L \rceil)$ 
13:         $b_k \leftarrow 0$ 
14:       $C_i \leftarrow i_1 b_1 i_2 b_2 \dots i_\Delta b_\Delta$  { new color }
15:       $L \leftarrow \Delta \lceil \lg L + 1 \rceil$ 

```

Theorem 4.1. The algorithm Color-Constant-Degree-Graph produces a valid coloring of a constant degree graph.

Proof. Each processor has a unique identification number, so the initial coloring is valid. We assume that the coloring at the beginning of an iteration is valid, and consider any two vertices, v and w . If v is not adjacent to w , then they may have the same

color. Therefore, the coloring is always valid for v and w . We are interested in the case v and w are two neighboring vertices. To obtain a contradiction, we assume that at the end of the iteration they have the same color, i.e., they have the same list with Δ pairs. Since v and w are two neighboring vertices, and the coloring at the beginning of an iteration is valid, there exists a minimum index i_k such that $C_v(i_k) \neq C_w(i_k)$. By the algorithm, at least one of the Δ pairs for C_v and C_w has index i_k . Since v and w have the same list at the end of the iteration, the values of the bit at i_k for C_v and C_w are equal, and this is a contradiction. ■

Lemma 4.1. $\lg \lceil x \rceil \leq \lceil \lg x \rceil$

Proof. If $2^{k-1} < x \leq 2^k$, then $\lg \lceil x \rceil = \lg 2^k = k = \lceil \lg x \rceil$.

If $2^{k-1} < x \leq 2^{k-1}$, then $\lg \lceil x \rceil \leq \lg (2^k - 1) < k = \lceil \lg x \rceil$.

Therefore, $\lg \lceil x \rceil \leq \lceil \lg x \rceil$. ■

Now we introduce some notation in order to analyze the algorithm, Color-Constant-Degree-Graph. Again $L = \lceil \lg n \rceil$, the number of bits in the representation of initial color, and also let L_k denote the number of bits in the representation of colors after k iterations.

Theorem 4.2. If $\lceil \lg^{(k)} L \rceil \geq \lceil \lg \Delta \rceil + 2$, then $L_k \leq 2\Delta \lceil \lg^{(k)} L \rceil$.

Proof. This can be shown by induction on k .

For $k = 1$, we have

$$\begin{aligned} L_1 &= \Delta \lceil \lg L + 1 \rceil \\ &\leq 2\Delta \lceil \lg L \rceil, \text{ because } \lceil \lg^{(1)} L \rceil \geq \lceil \lg \Delta \rceil + 2 \end{aligned}$$

Assume that for some $k - 1$, we have

$$L_{k-1} \leq 2\Delta \lceil lg^{(k-1)} L \rceil, \text{ if } \lceil lg^{(k-1)} L \rceil \geq \lceil lg \Delta \rceil + 2$$

Then

$$\begin{aligned} L_k &= \Delta \lceil lg L_{k-1} + 1 \rceil \\ &\leq \Delta (\lceil lg (2\Delta \lceil lg^{(k-1)} L \rceil) \rceil + 1) \\ &= \Delta (\lceil lg 2 + lg \Delta + lg \lceil lg^{(k-1)} L \rceil + 1 \rceil) \\ &\leq \Delta (\lceil \lceil lg \Delta \rceil + \lceil lg (lg^{(k-1)} L) \rceil + 2 \rceil) \quad , \text{ by Lemma 4.1} \\ &= \Delta (\lceil lg \Delta \rceil + 2 + \lceil lg^{(k)} L \rceil) \\ &\leq 2\Delta \lceil lg^{(k)} L \rceil \quad , \text{ if } \lceil lg^{(k)} L \rceil \geq \lceil lg \Delta \rceil + 2 \quad \blacksquare \end{aligned}$$

Theorem 4.3. Assume that $k = \min \{ i \mid \lceil lg^{(i)} L \rceil < \lceil lg \Delta \rceil + 2 \}$. Then the algorithm, Color-Constant-Degree-Graph, terminates after at most $k + c$ iterations, where c is a constant.

Proof. Since $\lceil lg^{(k-1)} L \rceil \geq \lceil lg \Delta \rceil + 2$, $L_{k-1} \leq 2\Delta \lceil lg^{(k-1)} L \rceil$ by the previous theorem. Then

$$\begin{aligned} L_k &= \Delta \lceil lg L_{k-1} + 1 \rceil \\ &\leq \Delta (\lceil lg (2\Delta \lceil lg^{(k-1)} L \rceil) \rceil + 1) \\ &= \Delta (\lceil lg 2 + lg \Delta + lg \lceil lg^{(k-1)} L \rceil + 1 \rceil) \\ &\leq \Delta (\lceil \lceil lg \Delta \rceil + \lceil lg (lg^{(k-1)} L) \rceil + 2 \rceil) \\ &\leq 2\Delta (\lceil lg \Delta \rceil + 2), \text{ since } \lceil lg^{(k)} L \rceil \leq \lceil lg \Delta \rceil + 2 \end{aligned}$$

Therefore, L_k is bounded above by the constant $2\Delta (\lceil lg \Delta \rceil + 2)$. By the algorithm, for any integer i , L_i is always greater than Δ , and for each iteration L_i decreases by at least Δ . Hence the algorithm iterates at most $k + 2(\lceil lg \Delta \rceil + 2)$ times. The desired conclusion follows. $\blacksquare$

Theorem 4.4. The algorithm, Color-Constant-Degree-Graph, colors a graph with a constant number of colors which is at most $2^{2\Delta(\lceil \lg \Delta \rceil + 2)}$.

Proof. We showed that $L_k \leq 2\Delta (\lceil \lg \Delta \rceil + 2)$ in the previous proof, where $k = \min \{ i \mid \lceil \lg^{(i)} L \rceil < \lceil \lg \Delta \rceil + 2 \}$. Since L_k denotes the number of bits in the representation of colors after k iterations, and this algorithm never increases the number of colors, it will terminate with a constant number of colors. ■

Corollary 4.1. The algorithm, Color-Constant-Degree-Graph, runs in $O(\lg^* n)$.

Proof. Each iteration of the algorithm, Color-Constant-Degree-Graph, uses $O(1)$ time. Thus by Theorem 4.3 the running time of the algorithm is $k + c$, where $k = \min \{ i \mid \lceil \lg^{(i)} L \rceil < \lceil \lg \Delta \rceil + 2 \}$, and c is a constant. Let's compare k with $\lg^* n$. Since $\lg^* n = \min \{ i \mid \lceil \lg^{(i)} n \rceil \leq 2 \}$, $k \leq \lg^* n$. Therefore, $k = O(\lg^* n)$, i.e., $k + c = O(\lg^* n)$. ■

The second algorithm, Max-Independent-Set, uses the algorithm Color-Constant-Degree-Graph to color the input graph with a constant number of colors. After the graph has been colored, the algorithm iterates over all the colors, selecting the remaining vertices of the current color, adding them to the independent set, and deleting all their neighbors from the graph.

Here, we need to show some set operations which are used in the following algorithms. Let U be a set. Then consider the following statement :

For each processor P_i , in parallel

$$U = U \cup \{i\}$$

Since we are dealing with a parallel algorithm, several processors try to update the set U at the same time. If we are using an ordinary set, this statement could not be implemented

efficiently. Therefore, we introduce an array U of Boolean type so that if i is in the set U , then $U[i]$ is True. Otherwise $U[i]$ is False. Now we can translate the above statement to the following :

For each processor P_i , in parallel

$U[i] = \text{True}$

Therefore, by using an array instead of a set, we can implement a set operation in constant time. However, we keep using the word "set" for convenience.

Algorithm Max-Independent-Set ($G, \text{Max-Ind-Set}$)

{ Input : a graph $G=(V,E)$ with constant maximum degree Δ , where $V = \{0,1,2,\dots,n-1\}$.

Output : Max-Ind-Set, a maximal independent set of G .

Uses n processors, $P_0, \dots, P_{n-1}$.

}

1: Color-Constant-Degree-Graph (G,L)

2: Max-Ind-Set $\leftarrow \emptyset$

3: for $k \leftarrow 0$ to $2^L - 1$

4: for each processor P_i in parallel

{ if vertex i is colored with k and is not adjacent to any
vertices already in Independent set, then add i into the set. }

5: if vertex i is colored with k and is not marked

6: Max-Ind-Set $\leftarrow \text{Max-Ind-Set} \cup \{i\}$

7: if vertex i is adjacent to a vertex which is in the Max-Ind-Set

8: mark i

Theorem 4.5. In a constant-degree graph, the algorithm Max-Independent-Set finds a maximal independent set in $O(lg^* n)$ time using $O(n)$ processors.

Proof. We first want to show that this algorithm returns a valid maximal independent set. For $(i+1)^{th}$ iteration, vertices colored with i are inserted into the set only if they are not adjacent to the vertices which are already in the set, and vertices colored with other than i are removed only if they are adjacent to the vertices which are in the set. Therefore no two vertices in the set are adjacent, and every removed vertex is adjacent to the vertex in the set.

The running time of this algorithm is dominated by the Color-Constant-Degree-Graph subroutine which takes time $O(lg^* n)$ by the Corollary 4.1: each iteration takes constant time, and the number of iterations is equal to the total number of colors which is 2^L , where $L \leq 2\Delta \lceil lg \Delta + 2 \rceil$, and therefore is a constant. ■

The third algorithm, $(\Delta+1)$ -Coloring, works by iteratively finding a maximal independent set Max-Ind-Set in the input graph, coloring vertices in Max-Ind-Set with a new color, and then deleting Max-Ind-Set from the graph.

Algorithm $(\Delta+1)$ -Coloring (G)

{ Input : a graph $G=(V,E)$ with constant maximum degree Δ , where $V = \{0,1,2,...,n-1\}$.

Output : a $(\Delta+1)$ coloring of G.

Uses n processors, $P_0, ..., P_{n-1}$.

}

```

1:   for  $k \leftarrow 1$  to  $\Delta+1$ 
2:       Max-Independent-Set ( $G$ , Max-Ind-Set)
3:       for each processor  $P_i$  in parallel do
4:           if  $i \in \text{Max-Ind-Set}$ 
5:               color  $i$  with  $k$ 
6:                $G \leftarrow G - \{i\}$ 

```

Theorem 4.6. A graph whose maximum degree is a constant Δ can be colored with $(\Delta+1)$ colors in $O(\lg^* n)$ time using $O(n)$ processors by the algorithm $(\Delta+1)$ -Coloring.

Proof. The $(\Delta+1)$ -Coloring algorithm works by iteratively finding a maximal independent set in the input graph, coloring vertices in the set with a new color, and then deleting all the vertices in the maximal independent set from the graph. If a vertex v is not removed at the end of some iteration, at least one of its neighbors is removed during this iteration. Therefore, after each iteration, the maximum degree of the graph induced by the remaining vertices decreases. Hence, after at most $\Delta+1$ iterations, the algorithm terminates with a $(\Delta+1)$ -coloring of the graph.

Since Δ is a constant, and removing a vertex from a graph G can be implemented in a constant time, the running time of this algorithm is determined by the Max-Independent-Set subroutine which takes time $O(\lg^* n)$. ■

Chapter 5

Parallel Δ -coloring algorithm

5.1 Introduction

We have already shown an efficient parallel $(\Delta+1)$ -coloring algorithm which generates a valid coloring for a given graph with the number of colors at most one greater than its maximum degree(Δ). However, Brook's theorem states that there exist colorings with fewer colors for most graphs.

Theorem (Brooks) A connected graph G of maximum degree Δ can be colored with Δ colors if and only if either $\Delta \geq 3$ and G is not the clique of $\Delta+1$ vertices, or $\Delta \leq 2$ and G is not an odd cycle.

The proofs of Brooks theorem, in Hakimi(unpublished) and Berge(1973), are sequentially algorithmic, but converting the Δ coloring sequential algorithms to parallel algorithms seems very difficult. Hajnal and Szemerédi(1990), Karloff(1989), and Karchmer and Naor(1988) independently announced parallel Δ -coloring algorithms. All three solutions used different techniques, but all of them are based on Luby's(1985) Maximal Independent Set Algorithm. In this chapter, we show the Hajnal-Szemerédi Algorithm which uses n^8 processors and runs in $O(lg^3 n)$ on CRCW PRAM. In this algorithm we will use Luby's Maximal Independent Set algorithm without discussing it further. We do note that the Luby's algorithm uses n^4 processors and runs in $O(lg^2 n)$ time.

We use the following notation and definitions throughout this chapter for convenience.

- 1) For any graphs G and H , $G + H$ denotes the graph formed by joining each vertex of G to each vertex of H .
- 2) For any given set C of colors, an allowable color $c \in C$ at vertex v is a color not used on v 's neighbors.
- 3) A partial coloring of G is an assignment of colors to some subset S of $V(G)$ such that each vertex of S has exactly one color and adjacent vertices of S have different colors.
- 4) A maximal k coloring of a graph G is a partial coloring of G such that if a vertex v in G is not assigned a color, then v is adjacent to vertices of every color in a color set C of size k .

Here, we assume that the given graph $G = (V, E)$, where the maximum degree is Δ and $V = \{1, 2, \dots, n\}$, is represented by an adjacency matrix. We also denote K_q as a complete graph on q vertices.

5.2 Sequential Δ Coloring

The Sequential- Δ -Coloring algorithm, presented in this section, is based on the proof of Brooks theorem presented by Hakimi(unpublished). Since his proof is sequentially algorithmic and a detailed explanation of a sequential algorithm is beyond the scope of this paper, we just present the required algorithm.

A connected graph G is in one of the following three categories:

1. G is not regular,
2. G is regular and contains a cut vertex v , and
3. G is regular and contains three vertices u , v , and w such that there are edges $\{u,v\}$ and $\{u,w\}$ in G , v is not adjacent to w , and $G - \{v,w\}$ is connected.

Therefore, the Sequential- Δ -Coloring algorithm also contains three main procedures, Irregular, Cut-Vertex, and Special, and each procedure works for its corresponding connected graph.

The first algorithm, Irregular, accepts either a connected graph G which is not regular and a vertex $v \in V(G)$ such that $\deg(v)$ is less than Δ , or a regular connected graph with a vertex v which has 2 non-adjacent neighbors already colored with 1. This algorithm outputs a coloring of G with Δ colors and takes $O(n^2)$.

Algorithm Irregular (G, v)

{ Input : a graph $G = (V,E)$ with maximum degree Δ , where $V = \{1,2,\dots,n\}$, and
a vertex $v \in V$ such that $\deg(v)$ is less than Δ or v has 2 non-adjacent neighbors
already colored with 1.

Output : a coloring of G with Δ colors.

Let C be the set of colors of size Δ .

}

- 1: Using Breadth First Search, find distances between the starting vertex v and other vertices, where distance is simply the number of edges in a shortest path.
- 2: Let Max be the maximum distance we just calculated

```

3:   while Max  $\geq$  0
4:       while there is an uncolored vertex w such that the distance between v and
           w is Max
6:           color w with the smallest allowable color in C
7:       Max = Max - 1

```

The second algorithm, Cut-Vertex, accepts a graph G which has a cut vertex v , and also outputs a coloring of G with Δ colors. Running time of this algorithm is $O(n^3)$.

Algorithm Cut-Vertex(G, v)

{ Input : a graph $G = (V, E)$ which has a cut vertex v .

Output : a coloring of G with Δ colors.

Let C be the set of colors of size Δ .

}

```

1:   Find components of  $G - \{v\}$ 
2:   Let Count be the number of components in  $G - \{v\}$ 
3:   for i = 1 to Count
4:       Let  $V_i$  be the set of vertices in the  $i^{\text{th}}$  component.
5:       run Irregular( $G(V_i \cup \{v\}), v$ )
6:       save color of  $v$  into an array Temp of size Count, and uncolor  $v$ 
7:   for i = 2 to Count
8:       if Temp[1]  $\neq$  Temp[i]
9:           exchange colors Temp[1] and Temp[i] in  $i^{\text{th}}$  component
10:  color  $v$  with Temp[1]

```


The third algorithm, Special, takes a input graph G , and finds a triple (u,v,w) such that there are edges $\{u,v\}$ and $\{u,w\}$ in G , v is not adjacent to w , and $G - \{v,w\}$ is connected. By the proof of Hakimi, such a triple always exists only if G is regular and does not contain a cut vertex. Then the algorithm assigns the same color to vertices v and w , and applies the algorithm Irregular on G and the vertex u . This algorithm also runs in $O(n^2)$.

Algorithm Special(G)

{ Input : a graph $G = (V,E)$ which is regular and does not contain a cut vertex.

Output : a coloring of G with Δ colors.

}

- 1: choose a vertex u and its two adjacent vertices v and w , where v is not adjacent to w .
- 2: if $G - w$ is connected, but has v as a cut vertex (i.e., $G - \{v,w\}$ is not connected.)
- 3: choose two vertices x and y such that both x and y are adjacent to w , they are not neighbors, and $G - \{x,y\}$ is connected. { Such vertices x and y always exist, by the proof of Hakimi. }
- 4: relabel (w,x,y) with (u,v,w) .
{ Now we have a triple (u,v,w) such that there are edges $\{u,v\}$ and $\{u,w\}$ in G , v is not adjacent to w , and $G - \{v,w\}$ is connected. }
- 5: color v and w with color 1.
- 6: run Irregular(G,u)

Now we have three main procedures so that only thing left is to put these three pieces together. The algorithm Sequential- Δ -Coloring does this job, running in $O(n^3)$.

Algorithm Sequential- Δ -Coloring (G)

{ Input : a graph G of maximum degree Δ , where either $\Delta \geq 3$ and G contains no clique of $\Delta+1$ colors, or $\Delta \leq 2$ and G has no odd cycle.

Output : a coloring of G with Δ colors.

}

- 1: if there exists a vertex v such that degree of v is less than Δ
- 2: run Irregular(G,v)
- 3: else if there exists a cut vertex v
- 4: run Cut-Vertex (G,v)
- 5: else run Special (G)

5.3 Parallel Δ Coloring

Before explaining the Hajnal-Szemerédi Algorithm, we show a simple statement which works in parallel and finds an allowable color set C_v for each vertex v , using n^2 processors and running in constant time. We assume that we have a given color set C and we use CRCW PRAM. Then the statement is as follows :

for each processor P_{ij} , in parallel

$$C_i \leftarrow C$$

if a colored vertex j is adjacent to a vertex i

$$C_i \leftarrow C_i - \{\text{color of } j\}$$

By using an array of Boolean type as a set, we can implement the above set operation in parallel. (This has been shown in Chapter 4.)

The Hajnal-Szemerédi Algorithm starts by finding a maximal $(\Delta - 3)$ coloring of a given graph $G=(V,E)$, using Luby's(1985) algorithm. If we let U be the set of uncolored vertices in G , then the degree of an uncolored vertex in $G(U)$ is at most 3. The algorithm then repeatedly recolors G until $G(U)$ does not contain any K_4 's, but we still have maximal $(\Delta - 3)$ coloring of G . If $G(U)$ does not contain any K_4 's, then the algorithm of Karloff(1985), an algorithm for 3-coloring a maximum degree 3 graph without K_4 's, can be applied to complete the Δ coloring of G .

In the next section we show Karloff's Algorithm which colors a graph $G = (V,E)$ with only three colors, where the maximum degree of G is at most three and G is not the clique of 4 vertices. In sequent sections we consider the Hajnal-Szemerédi Algorithm which colors a graph $G = (V,E)$ with Δ colors, where G is not the clique of $(\Delta+1)$ vertices and Δ is at least 4.

5.3.1 Karloff's Algorithm

In this section, we show the algorithm of Karloff(1985) which colors a graph $G = (V, E)$, where $1 \leq \Delta(G) \leq 3$ and no component of G is a K_4 .

Before explaining Karloff's algorithm, we need to show a preliminary algorithm, named Grouping. Let a graph G have a maximum degree at most 2. When we define an odd cycle as a cycle with odd number of vertices and an even cycle as a cycle with even number of vertices, each component of G then falls into one of the four groups : isolated vertex, path, even cycle, or odd cycle. The preliminary algorithm Grouping places each component into its appropriate group.

Algorithm Grouping (G)

{ Input : a graph G with maximum degree less than or equal to 2.

Output : grouping of each component of G .

Uses n^4 processors and runs in $O(\lg n)$ (determined by the algorithm All-Pairs'-Distances. Also line 15 uses $O(\lg n)$ time.)

D : array of size $|V(G)| = n$, which is an output of algorithm Components

M : $n \times n$ matrix, which is an output of algorithm All-Pairs'-Distances

Group : array of size n , which denotes a group which each vertex belongs to.

Cycles : a set of vertices which are in cycles, initially empty.

}

```

1:   Run Components and save its output in D
    { Initially we assume we have only cycles. }
2:   for each processor  $P_{i111}$ , in parallel
        Group[i] = cycle
    { Grouping of isolated vertices }
3:   for each processor  $P_{ij11}$ , in parallel
4:       if  $\deg(i) = 0$    { Processor  $P_{i11}, \dots, P_{in}$  are used to decide if  $\deg(i) = 0$ . }
5:           Group[i] = isolated
    { Grouping of path }
6:   for each processor  $P_{ij11}$ , in parallel
7:       if  $\deg(i) = 1$    { Processor  $P_{i11}, \dots, P_{in}$  are used to decide if  $\deg(i) = 0$ . }
8:           Group[D(i)] = path
    { Grouping of odd cycle and even cycle }
9:   for each processor  $P_{i111}$ , in parallel
10:      if Group[i] is Cycle
11:          Cycles = Cycles  $\cup$  {i}
12:  Run All-Pairs'-Distances on  $G(\text{Cycles})$  and save its output on M
13:  for each processor  $P_{ij11}$ , in parallel
14:      if  $D(i) = i$  and Group[i] is Cycle
          { Now n processors,  $P_{i111}, P_{i211}, \dots, P_{in11}$ , work to find the values
            we require }
15:          find the first and second maximum distances on  $i^{\text{th}}$  row of M.
16:          if those distances are the same
17:              Group[i] = Odd
18:          else   Group[i] = Even

```

19: for each processor P_{i111} , in parallel

20: Group(i) = Group(D(i))

Since an input graph G of Karloff's algorithm has a maximum degree at most 3, we see that a maximum degree of $G - I$ is at most 2, for a maximal independent set I of G . Therefore, by the algorithm Grouping, we can place each component of $G-I$ into its appropriate group. Assuming that $G-I$ does not contain an odd cycle, we can easily color G with 3 colors (c_1, c_2, c_3) by the following four steps:

1. Apply algorithm All-Pairs'-Distances on $G-I$, assuming all edges have weight 1,
2. Choose a vertex v in each component in $G-I$ and assign a color c_1 to v and to every vertex u , where the shortest path uv has even length,
3. Assign a color c_2 to all uncolored vertices in $G-I$, and
4. Assign a color c_3 to all vertices in I .

Therefore, if we find a maximal independent set I' of G such that $G-I'$ does not contain any odd cycles, we are done. Assume that for a maximal independent set I of G , $G-I$ has k odd cycles and there is an algorithm A which uses the set I and generates a new maximal independent set I' of G such that the number of odd cycles in $G-I'$ is at most $\lfloor k/3 \rfloor$. Then by running the algorithm A $\lceil \log_3 n \rceil$ times, we can have a maximal independent set I' such that no odd cycles exist in $G-I'$. Now we show such algorithm, named NewIndep.

The algorithm NewIndep accepts as an input a maximal independent set I of G , where $1 \leq \Delta(G) \leq 3$ and no component of G is a K_4 , and outputs a new maximal independent set I' of G such that the number of odd cycles in $G-I'$ is at most $1/3$ times that in $G-I$. Before giving the detailed algorithm NewIndep, we describe its major steps.

- step 1. Let $I' = \emptyset$
- step 2. $I' = I' \cup \{\text{all isolated vertices in } G-I\}$
- step 3. For each path P in $G-I$, add to I' one end vertex v of P and all vertices of P at even distance along P from v . Each end vertex not in I' is now called the unmarked end vertex of P .
- step 4. For each even cycle C in $G-I$, add alternate vertices from C to I' , starting at any vertex.
- step 5. For each odd cycle C in $G-I$, choose a consecutive triple of vertices $\{a_c, b_c, c_c\}$, where a_c 's neighbor in I is different from c_c 's. Such a triple always exists, since $\Delta(G) \leq 3$ and no component of G is a K_4 .
- step 6. Let $J = \{\text{unmarked end vertices of paths in } G-I\}$
 Let $L = J \cup \{b_c : C \text{ is an odd cycle in } G-I\}$
- step 7. $I' = I' \cup \{v : v \in I, \text{ and all neighbors of } v \text{ are in } L\}$
- step 8. For each odd cycle C in $G-I$, find a critical pair, either $\{a_c, b_c\}$ or $\{b_c, c_c\}$.
 Let x, y, z be the neighbors in I of a_c, b_c, c_c , respectively.
- Case1. If $x = y$, then do :
- If degree of x in G is 2 or x has a neighbor in J , then specify $\{a_c, b_c\}$ as C 's critical pair, and add x to I' .
 Otherwise specify $\{b_c, c_c\}$ as C 's critical pair.
- Case 2. If $z = y$, then do :
- If degree of z in G is 2 or z has a neighbor in J , then specify $\{b_c, c_c\}$ as C 's critical pair, and add z to I' .
 Otherwise specify $\{a_c, b_c\}$ as C 's critical pair.

Case 3. If $x \neq y \neq z$, and all neighbors of x other than a_c are in J or degree of x in G is 1,

then add x to I' , and specify $\{a_c, b_c\}$ as C 's critical pair.

Case 4. If $x \neq y \neq z$, and all neighbors of z other than c_c are in J or degree of z in G is 1, and case 3 doesn't hold,

then add z to I' , and specify $\{b_c, c_c\}$ as C 's critical pair.

Case 5. Otherwise, i.e., if $x \neq y \neq z$, and both x and z have neighbors other than a_c, c_c which are not in J ,

Let H_c be the subgraph of G induced by $(I \cup \{a_c, b_c, c_c\} \cup J) - I'$.

Let $H_c^a = H_c - \{c_c\}$, and $H_c^c = H_c - \{a_c\}$

We prove in lemma 5.1 that both H_c^a and H_c^c have maximum degree at most two.

If the component of H_c^a containing b_c is a path, then specify $\{a_c, b_c\}$ as C 's critical pair.

Otherwise, specify $\{b_c, c_c\}$ as C 's critical pair.

step 9. For each odd cycle $C(v_1, v_2, v_3, \dots, v_{2k}, v_{2k+1}, v_1)$, let v_1 and v_2 be the critical pair. Then add $v_3, v_5, \dots, v_{2k+1}$ to I' .

step 10. Return extension of I' to a maximal independent set.

We note that if vertices u and v are in the same set, I or J , then u is not adjacent to v . Also note that an odd cycle C in $G - I'$ can contain only vertices of I , J , and critical pairs, since other vertices have at least two neighbors in I' or at most one neighbor in $G - I'$. Our goal is to show that every odd cycle in $G - I'$ contains at least three critical pairs. Theorem 5.1 shows that no odd cycle contains exactly 2 or 0 critical pairs, and theorem 5.2 shows that there is no odd cycle in $G - I'$ containing only one critical pair.

Theorem 5.1. No odd cycle of $G-I'$ contains exactly 0 or 2 critical pairs.

Proof. Assume that a cycle C contains no critical pair. Then the cycle C contains vertices only of I and J . Therefore, the cycle C has the following format : $i_1 e_1 \dots i_p e_p \dots e_{n-1} i_n e_n i_1$, where $i_k \in I$ and $e_k \in J$. Hence the cycle C can not be an odd cycle. Assume that a cycle C contains exactly two critical pairs. Since there is no critical pair adjacent to a vertex in J , the cycle C contains the following format : $i_1 e_1 \dots e_{p-1} i_p$ (critical pair) $i_{p+1} e_p \dots e_{q-2} i_q$ (critical pair) $i_{q+1} e_{q-1} \dots i_n e_{n-2} i_1$, where $i_k \in I$ and $e_k \in J$. Therefore the cycle C can not be an odd cycle. ■

Lemma 5.1. In case 5 of step 8, Both H_c^a and H_c^c have maximum degree at most 2.

Proof. Note that $V(H_c) = (I \cup \{a_c, b_c, c_c\} \cup J) - I'$. We first consider H_c^a . Both a_c and b_c obviously have degree 2. Assume v was in J . By step 3, one of its neighbors must be in I' . Therefore, v has maximum degree at most 2 in H_c^a . Assume v was in I . If v is adjacent to both a_c and b_c , this case would have been considered by case 1 and v is in I' . If v is adjacent to either a_c or b_c , in order not to be covered by case 3 and step 7, v should have at most only one neighbor in J , i.e., v has maximum degree at most 2. If v is adjacent to vertices only in J and has degree 3, then by step 7, v is in I' . Hence, H_c^a has maximum degree at most 2. Similarly H_c^c also has maximum degree at most 2. ■

Lemma 5.2. In case 5 of step 8, either the component of b_c in H_c^a is a path, or the component of b_c in H_c^c is a path.

Proof. Note that both H_c^a and H_c^c have maximum degree at most 2. Let x, y, z be the neighbors in I of a_c, b_c, c_c , respectively, and suppose that the component A of H_c^a which contains b_c is a cycle. Then the cycle A has the following sequence, necessarily

of odd length : $b_c, y, e_1, i_1, \dots, e_{p-1}, i_{p-1}, e_p, x, a_c, b_c$, where $i_k \in I$ and $e_k \in J$. Vertex z is not in the sequence because if so, z has two neighbors in J and so this is case 4. Since both H_c^a and H_c^c have maximum degree at most 2, and H_c^c differs from H_c^a only in that a_c has been deleted and c_c added, both have the same sequence, $b_c, y, e_1, i_1, \dots, e_{p-1}, i_{p-1}, e_p, x$. But in H_c^c , vertex x has only one neighbor e_p , and so x must be an end vertex in the component of b_c . Therefore, the component of b_c in H_c^c is a path. ■

Theorem 5.2. After NewIndep halts, there is no odd cycle in $G-I'$ containing only one critical pair.

Proof. Note that an odd cycle C in $G-I'$ can contain vertices only of I, J , and critical pairs, and also note that any two vertices in the same set, either I or J , are not adjacent to each other. Therefore, we may say that an odd cycle in $G-I'$ containing only one critical pair has the following format : $i_1 e_1 \dots i_p$ (critical pair) $i_{p+1} \dots i_n e_n i_1$, where $i_k \in I$ and $e_k \in J$. A critical pair is chosen by five distinct cases of step 8. In order to show that no critical pair chosen by any of these five cases is in an odd cycle in $G-I'$ containing only one critical pair, we consider each case.

In Case 1, such critical pair is not chosen. If $\{a_c, b_c\}$ has been chosen as a critical pair, all their neighbors will be in I' . If $\{b_c, c_c\}$ has been chosen, y in I has no neighbor in J

Case 2 is similar to case 1.

In Case 3, such critical pair is not chosen. Since x is added into I' , a_c has only one neighbor b_c , and so a_c is an end vertex of that component in $G-I'$.

Case 4 is similar to case 3.

In Case 5, such critical pair is not chosen. By lemma 5.2, this is obvious. ■

Theorem 5.3. When NewIndep halts, the number of odd cycles in $G-I'$ is at most one third the number of odd cycles in $G-I$.

Proof. By theorem 5.1 and 5.2, an odd cycle C of $G-I'$ contains at least three critical pairs of $G-I$. We note that every odd cycle in $G-I$ has a critical pair and no critical pair belongs to two different odd cycles of $G-I'$ because maximum degree of $(G-I')$ is at most 2. Consequently the number of odd cycles in $G-I'$ is at most one third the number of odd cycles in $G-I$. ■

Algorithm NewIndep(G, I)

{ Input : a graph G and a maximal independent set I of G , where $\Delta(G)$ is at most 3 and no component of G is a K_4

Output : a new maximal independent set I' of G such that the number of odd cycles in $G-I'$ is at most $1/3$ of the number of odd cycles in $G-I$

Uses n^4 processors and runs in $O(\lg n)$ (determined by the algorithm All-Pairs'-Distances.)

Critical : array of size n , whose non-zero entries are vertices b_c , when $\{a_c, b_c, c_c\}$ is a triple chosen by step 5.

Critical-Pair : array of size n , whose non-zero entries are critical pairs of odd cycles.

EndVx : array of size n , whose non-zero entries are end vertices of paths.

We assume that we keep an n vector D , which is an output of algorithm Components, where $D(v) = D(w)$ if and only if vertex v is in the same component as vertex w . And also we keep an $n \times n$ matrix M , which is an output of algorithm All-Pairs'-Distances, where M_{ij} is a distance between i and j .

}

{ step 1 : Initialization. I' , J , and L will be used as discussed earlier, and Y becomes a set of vertices, some of which are added to I' at the end of the algorithm.}

1: $I' \leftarrow \emptyset, J \leftarrow \emptyset, L \leftarrow \emptyset, Y \leftarrow \emptyset$

{ step 2 : If component is an isolated vertex }

2: for each processor P_{i111} , in parallel

3: if vertex i is isolated in $G-I$

4: $I' \leftarrow I' \cup \{ i \}$

{ step 3 : If component is a path }

5: for each processor P_{i111} , in parallel

6: if vertex i is in a path and $\deg(i) = 1$ in $G-I$

7: $\text{EndVx}[D(i)] = i$ { requires priority write PRAM }

8: for each processor P_{i111} , in parallel

9: if vertex i is in a path

10: if $M_{i\text{EndVx}[D(i)]}$ is even.

11: $I' \leftarrow I' \cup \{ i \}$

12: if i has degree 1 in $G-I$ but $i \notin I'$

13: $J \leftarrow J \cup \{ i \}$

{ step 4 : If component is an even cycle }

14: for each processor P_{ij11} , in parallel

15: if vertex i is in an even cycle and $M_{D(i)j}$ is even

16: $I' \leftarrow I' \cup \{ j \}$

{ If component is an odd cycle }

17: for each processor P_{i111} , in parallel

18: if vertex i is in odd cycle C

19: Let a and c in C be the two adjacent vertices of i , and also let x, y, z
 be the neighbors in I of a, i, c , respectively.
 { step 5 : find consecutive triple $\{a, i, c\}$, where a 's neighbor in I is
 different from c 's. }

20: if $x \neq z$

21: $\text{Critical}[D(i)] = i$ { requires priority write PRAM }
 { step 6 }

22: $L \leftarrow J \cup \{ \text{Critical}[D(i)] \}$
 { step 7 }

23: for each processor P_{ij11} , in parallel

24: if vertex i is in I and all neighbors of i are in L

25: $I' \leftarrow I' \cup \{ i \}$

26: for each processor P_{ij11} , in parallel

27: if vertex i is in odd cycle C and $\text{Critical}[D(i)] = i$

28: Let a and c in C be the two adjacent vertices of i , and also let x, y, z
 be the neighbors in I of a, i, c , respectively.
 { Case 1 : If $x = y$ }

29: if $x = y$

30: if degree of x in G is 2 or x has a neighbor in J

31: $\text{Critical-Pair}[D(i)] = (a, i)$

32: $I' \leftarrow I' \cup \{ x \}$

33: else $\text{Critical-Pair}[D(i)] = (i, c)$
 { Case 2: If $y = z$ }

34: else if $y = z$

35: if degree of z in G is 2 or z has a neighbor in J

```

36:           Critical-Pair[D(i)] = (i,c)
37:           I' ← I' ∪ { z }
38:       else    Critical-Pair[D(i)] = (a,i)
               { Case 3 : If  $x \neq y \neq z$ , and all neighbors of  $x$  other than  $a$  are in  $J$ 
                 or degree of  $x$  in  $G$  is 1 }
39:       else if degree of  $x$  in  $G$  is 1 or all neighbors of  $x$  other than  $a$  are in  $J$ 
40:           Critical-Pair[D(i)] = (a,i)
41:           I' ← I' ∪ { x }
               { Case 4 : If  $x \neq y \neq z$ , and all neighbors of  $z$  other than  $c$  are in  $J$ 
                 or degree of  $z$  in  $G$  is 1 }
42:       else if degree of  $z$  in  $G$  is 1 or all neighbors of  $z$  other than  $c$  are in  $J$ 
43:           Critical-Pair[D(i)] = (i,c)
44:           I' ← I' ∪ { z }
               { Case 5 : Otherwise }
45:       else    Let  $H$  be the subgraph of  $G$  induced by  $(I \cup \{a,i,c\} \cup J) - I'$ .
46:               Let  $H_a = H - \{c\}$ , and  $H_c = H - \{a\}$ 
47:               Run Components on  $H_a$ 
48:               if the component of  $H_a$  containing  $i$  is a path
49:                   Critical-Pair[D(i)] = (a,i).
50:               else    Critical-Pair[D(i)] = (i,c)
               { step 9 . Now we have a critical pair for each odd cycle}
51:       Let Critical-Pair[D(i)] = (a,b)
52:       Let  $C$  be the odd cycle containing  $i$ , and  $C' = C - \{a\}$ 
53:       Run All-Pairs'-Distances on  $C'$ 
54:       if a distance between  $b$  and  $j$  is odd

```

```

55:            $I' \leftarrow I' \cup \{j\}$ 
56:   for each processor  $P_{i1}$ , in parallel
57:       if vertex  $i$  is not in  $I'$  and no neighbors of  $i$  are in  $I'$ 
58:            $Y \leftarrow Y \cup \{i\}$ 
59:   Find a maximal independent set  $Y$  on  $G(Y)$ 
60:    $I' \leftarrow I' \cup Y$ 
61:   Return  $I'$ 

```

By the algorithm NewIndep, we are able to get a maximal independent set I of G such that $G-I$ contains no odd cycle. Therefore, the only remaining task is to give a coloring.

Algorithm Karloff

{ Input : a graph G , where $\Delta(G)$ is at most 3 and no component of G is a K_4

Output : a coloring of G with 3 colors, $\{c_1, c_2, c_3\}$

Uses n^4 processors and runs in $O(\lg^2 n)$

Global memories used

$n \times n$ matrix M : output of algorithm All-Shortest-Paths

n vector D : output of algorithm Components

}

{ Color isolated vertices in G with color c_1 }

```

1:   for each processor  $P_{i111}$ , in parallel
2:       if vertex  $i$  is isolated in  $G$ 

```

```

3:          color i with  $c_1$ 
4:           $G = G - \{i\}$ 
5:      Run Luby's algorithm to find a maximal independent set I of G
6:      Run Components on G-I
7:      Run All-Pairs'-Distances on G-I
8:      Run Grouping on G-I
      { Find the maximal independent set I, where no components of G-I are odd cycles}
9:      while there exists an odd cycle in G-I
10:         I = NewIndep(I)
11:         Run Components on G-I
12:         Run All-Pairs'-Distances on G-I
13:         Run Grouping on G-I
         { Color each vertex in the set I with color  $c_1$  }
14:     for each processor  $P_{i111}$ , in parallel
15:         if vertex i is in I
16:             Color i with  $c_1$ 
17:     Run All-Pairs'-Distances on G-I
     { Color vertices in G-I with two colors,  $c_2$  and  $c_3$  }
18:     for each processor  $P_{ij11}$ , in parallel
         { color isolated vertices with  $c_2$ }
19:         if vertex i is an isolated vertex
20:             color i with  $c_2$ 
         { color vertices in a path with  $c_2$ }
21:         if vertex i is in a path and  $D(i) = i$ 

```



```

22:          color EndVx[i] with  $c_2$           { EndVx is a global variable used by
                                                NewIndep algorithm. }
23:          if  $M_{\text{EndVx}[i]j}$  has even length
24:              color j with  $c_2$ 
                { color vertices in a cycle with  $c_2$  }
25:          else if vertex i is in an even cycle and  $D(i) = i$ 
26:              color i with  $c_2$ 
27:          if  $M_{ij}$  is even
28:              color j with  $c_2$ 
                { color all uncolored vertices with  $c_3$  }
29:  for each processor  $P_{i111}$ , in parallel
30:      if vertex i is not assigned a color
31:          color i with  $c_3$ 

```

Theorem 5.4. Karloff's algorithm runs in $O(lg^2 n)$, using n^4 processors.

Proof. The "while" loop in Karloff's algorithm iterates at most $\lceil \log_3 n \rceil$ times, and the procedure Components and All-Pairs'-Distances take $O(lg n)$ time. Therefore, the "while" loop takes $O(lg^2 n)$ time which is equal to the running time of Luby's algorithm to find a maximal independent set on G . The number of processors is determined by Luby's algorithm and All-Pairs'-Distances both of which use n^4 processors. Therefore, the running time of Karloff's algorithm is in $O(lg^2 n)$, using n^4 processors. ■

5.3.2 Algorithm Extend

The Extend algorithm accepts as an input a subset S of $V(G)$, where some of the vertices may be already colored, and finds a maximal $(\Delta-3)$ coloring of G on the set S . We have already shown that we can easily get an allowable color set C_v for each vertex v in S . By using these color sets C_v and the input S , we form a new graph $G' = (V', E')$ which has the following properties:

1. $V' = \{(v, c) : \text{uncolored } v \in S \text{ and } c \in C_v\}$ and
2. Two vertices $(v, c_1), (w, c_2)$ of V' are adjacent in G' if and only if
either $v = w$ and $c_1 \neq c_2$, or $c_1 = c_2$ and $\{v, w\} \in E(G)$

Now the algorithm finds a maximal independent set I of G' and then colors a vertex v with color c only if $(v, c) \in I$.

Algorithm Extend(S)

{ Input : a subset $S \subseteq V(G)$, where some of the vertices may be already colored.

Output : a maximal $(\Delta-3)$ coloring of G on the set S .

Uses n^8 processors for step 13. Other steps use n^4 processors, P_{ijkl} for $1 \leq i, j, k, l \leq n$.

The running time is in $O(lg^2 n)$.

Global memories used:

V' is a set of ordered pairs (i, j) where i is a vertex and j is an allowable color for i .

V' will be used as a vertex set of a new graph G' .

E' is a new set of edges.

I is a maximal independent set of the new graph G' .

Let $C = \{1, 2, \dots, \Delta-3\}$, and C_i be the set of allowable colors of vertex i .

}

```

1:   S ← S - Colored Vertices
2:   V' ← ∅
3:   E' ← ∅
      { Build a set V' of vertices }
4:   for each processor Pij11, in parallel
5:       if vertex i ∈ S and color j ∈ Ci
6:           V' ← V' ∪ {(i,j)}
      { Build a set E' of edges }
7:   for each processor Pijkl, in parallel
8:       if (i,j) and (k,l) are in V'
9:           if (i = k) and (j ≠ l)
10:              E' ← E' ∪ {(i,j),(k,l)}
11:              if (i ≠ k) and (j = l) and (i,k) ∈ E(G)
12:                  E' ← E' ∪ {(i,j),(k,l)}
13:   Find a maximal independent set I of G' = (V',E'), using Luby's algorithm.
14:   for each processor Pij11, in parallel
15:       if (i,j) ∈ I
16:           color i with j

```

Theorem 5.5. Let $G=(V,E)$ be a partially colored graph, $C = \{1,...,\Delta-3\}$ be a set of colors, and also let $S \subseteq V$. Then after executing the algorithm Extend on S , a vertex $v \in S$ is either colored $c \in C$ and not adjacent to any vertex in $V(G)$ colored c , or uncolored and adjacent to every color in C .

Proof. Assume that after executing the algorithm, vertices v and u are colored c , and adjacent to each other. Since G was a partial colored graph, at least one of them was just colored c . If either one of the vertices, say v , was not in $G(S)$, then v was already colored c because this algorithm assigns a color to a vertex only in $G(S)$. Therefore, vertex u has been colored by this algorithm. However, this cannot happen, since color c was not an allowable color of v , i.e., $(v,c) \notin V'$. If both v and u were in $G(S)$, then color c should be an allowable color of both vertices in order that both are colored c . However, since v and u are adjacent to each other and have allowable color c , edge $\{(v,c),(u,c)\}$ is in E' , i.e., both (v,c) and (u,c) cannot be placed in the maximal independent set I at the same time.

Now we want to show that an uncolored vertex v in S is adjacent to every color in C . Assume that the uncolored vertex v has an allowable color c after algorithm Extend terminates. Then $(v,c) \notin I$. Thus in G' , (v,c) is adjacent to a vertex $(v',c') \in I$. By the definition of G' , we have only two possible cases :

1. $v = v'$ and both c and c' are available at v , or
2. $c = c'$ and $\{v,v'\} \in E(G)$.

In the former case v is colored c' , and in the latter case v' is colored with c , i.e., c is not an allowable color for v . Therefore, in either case we have a contradiction. ■

5.3.3 Algorithm Eliminate

Assume that we have a maximal $(\Delta - 3)$ coloring of G , and let U be the set of uncolored vertices in G . If $G(U)$ does not contain any K_4 's, we can color $G(U)$ with only three colors, using Karloff's algorithm. Now our job is to get rid of each uncolored K_4 by coloring one, say v , of its vertices. It is essential that one of v 's colored adjacent vertices must be newly uncolored at the same time in order that the partial coloring of G remains valid. However if we have a K_4 in $G(U)$ all of whose vertices have the same colored neighbors, then this method may get stuck in an infinite loop. We will call such K_4 a bad K_4 , and make the following definition;

Let K denote an uncolored K_4 .

Then we say that K is a bad K_4 if and only if

$$\text{for all } v, u \in V(K), CN_v = CN_u,$$

where $CN_i = \{\text{colored vertices adjacent to a vertex } i\}$.

If a K_4 is not bad then we say that it is good. Figure 5.1 shows examples of bad K_4 's.

We note that if K is a bad K_4 , then its common colored neighbors contain two non-adjacent vertices, say u, v , as otherwise G would have $K_{\Delta+1}$ as a subgraph. The algorithm Eliminate first finds such vertices u and v . Then select any two vertices x, y from the K_4 and shift colors of u and v to x and y , respectively. Then apply algorithm Extend so that a new bad K_4 is not generated.

H : $\overset{1}{\bullet} \quad \overset{2}{\bullet} \quad \dots \quad \overset{\Delta-3}{\bullet}$

+

K_4 :

(a) A bad K_4 with H inducing as few edges as possible.

H : $\overset{1}{\bullet} + \overset{2}{\bullet} + \dots + \overset{\Delta-4}{\bullet} \quad \overset{\Delta-3}{\bullet}$

+

K_4 :

(b) A bad K_4 where H induces $K_{\Delta-4} \cup K_1$.

Figure 5.1. Examples of bad K_4 's. Each solid vertex has a labeled color.

Algorithm Eliminate (G)

{ Input : a maximal $(\Delta-3)$ coloring of G.

Output : a maximal $(\Delta-3)$ coloring of G without any bad K_4 's.

The number of processors and the running time is dominated by step 21 ~ 23 which call the algorithm Extend. Other steps use n^3 processors, P_{ijk} for $1 \leq i, j, k \leq n$, and run in $O(\lg n)$.

Global memories used :

U and U' are sets of uncolored vertices.

M is a connectivity matrix of $G(U)$

For $1 \leq i \leq n$,

bad K_4 [i] and good K_4 [i] determines if a K_4 is bad or good.

Non-Adj[i] holds a pair (u,v) of colored vertices such that u are not adjacent to v,
and both u and v are adjacent to all vertices of a bad K_4 .

String[i] is a set of vertices in the same component of $G(U)$.

```

}
1:  Let U be the set of uncolored vertices.
2:   $U' = \{ \}$ 
3:  Find a connectivity matrix M of  $G(U)$ 
4:  for each processor  $P_{ij1}$  in parallel
5:      if vertex i doesn't have a color
6:          if  $M_{ij} = 1$ 
7:               $String[i] = String[i] \cup \{j\}$ 
8:              if String[i] induces a  $K_4$  { requires constant time }
9:                  if  $K_4[i]$  is bad { We use processors  $P_{i1}, \dots, P_{in}$  to check this. }
10:                      $badK_4[i] = String[i]$ 
11:                     else  $goodK_4[i] = String[i]$ 
12:  for each processor  $P_{ijk}$  in parallel
13:      if i is the first character in  $badK_4[i]$  { this comparison is simple since we
                                         use CRCW PRAM. }
14:          if vertices j and k are colored neighbors of i, and these are not
          adjacent to each other { such vertices exist for otherwise G is
          complete }
15:              Non-Adj[i] = (j,k)

```

```

16:   for each processor  $P_{i11}$  in parallel
17:       if vertex  $i$  is in a  $\text{bad}K_4$ 
18:            $U' = U' \cup \{u,v\}$ , where  $\text{Non-Adj}[i] = (u,v)$ 
19:           choose any two vertices in  $\text{bad}K_4[i]$ , say  $x$ , and  $y$ 
20:           color  $x$  with color of  $u$  and  $y$  with color of  $v$ 
21:           uncolor  $u$  and  $v$ .
22:   Extend( $\cup \text{good}K_4[i]$ )
23:   Extend( $U'$ )
24:   Extend( $V(G)$ )

```

Theorem 5.6. When algorithm Eliminate terminates, the $(\Delta-3)$ -coloring of G is maximal, every uncolored K_4 was also uncolored when procedure Eliminate was called, and there are no bad K_4 's.

Proof. Since the algorithm concludes with 3 calls to Extend and the coloring is valid when the procedure Extend is called, it is obvious that the coloring of G is maximal. We denote that CN_K is the common colored neighbors of vertices of K , where K is a bad K_4 . Assume that H is currently an uncolored K_4 that was not completely uncolored when procedure Eliminate was called. Then there is a vertex $v \in H$ which lost its color during execution of line 21 of procedure Eliminate.

Case (1) The vertex v was an element of $CN_{K'}$ and $CN_{K''}$, where K' and K'' were two bad K_4 's. Since v is adjacent to every vertex in K' and every vertex in K'' , after executing line 20, v is adjacent to at least 4 uncolored neighbors none of which is colored in line 22. Therefore, line 21 colors the vertex v , and this is a contradiction.

Case (2) The vertex v was an element in only one CN_K , say $CN_{K'}$. Then after executing line 20, v has exactly two uncolored neighbors in K' . Thus there is a newly uncolored vertex u which creates the H together with v and the two uncolored vertices in K' . Since u is newly uncolored, u was an element of $CN_{K''}$, where K'' was a bad K_4 . If $K' \neq K''$, u is also adjacent to two uncolored vertices in K'' . Therefore, u has 4 uncolored neighbors and line 22 will color u . If $K' = K''$, v and u are the vertices chosen by line 15 and uncolored by line 21. However, by line 15, v and u are not adjacent to each other, i.e., the vertex v cannot create a new K_4 together with the vertex u . Therefore, every uncolored K_4 was also uncolored when procedure Eliminate was called.

Now we want to show that after procedure Eliminate terminates, there are no bad K_4 's. Since the algorithm does not create a new K_4 and all bad K_4 's which existed when the procedure was called are eliminated, all the currently uncolored K_4 's were good. Thus it is enough to show that the corresponding colored neighbors were not changed. Let K be a good K_4 when the procedure Eliminate was called. If any colored vertex adjacent to a vertex v of K is uncolored during the procedure, the vertex v would be colored by line 22, and K would be eliminated. Otherwise, K remains as a good K_4 . ■

5.3.4 Algorithm Modify-Coloring

To explain Modify-Coloring Algorithm which runs right after the algorithm Eliminate, we introduce some notation:

1. U and U' are the sets of uncolored vertices at the beginning of this algorithm and at the end of this algorithm, respectively,
2. C_i , $i = 1, \dots, \Delta-3$, is the set of vertices colored i ,
3. $Q = \{K_4\text{'s in } G(U)\}$,
4. $T = \{K_3\text{'s that are components in } G(U)\}$, and
5. R is the subset of uncolored vertices lying in components that are neither K_4 's nor K_3 's.

Since by Theorem 5.6, the output of the algorithm Eliminate is a maximal $\Delta-3$ coloring of G , every vertex v of degree 3 in $G(U)$ has exactly one neighbor in every C_i . For every vertex v of degree two in $G(U)$, there are two possibilities: v has exactly one neighbor in every C_i or there is exactly one C_i with two neighbors of v and each C_j other than C_i has exactly one neighbor of v . Therefore, the class T can be partitioned into $\Delta-2$ classes: for $1 \leq i \leq \Delta-3$, $T_i = \{K_3\text{'s in } T \text{ such that each node in the } K_3 \text{ has two neighbors colored } i\}$, and $T_0 = T - (T_1 \cup T_2 \cup \dots \cup T_{\Delta-3})$. We say that Modify-Coloring creates a recoloring problem when it creates one or more new K_4 's in $G(U')$ while eliminating one K_4 in $G(U)$.

Modify-Coloring Algorithm accepts as an input a graph G of maximum degree Δ and a maximal $\Delta-3$ coloring, where all K_4 's in $G(U)$ are good, and recolors the graph G so that the number of K_4 's in $G(U')$ is at most $3/4$ times the number of K_4 's in $G(U)$. This algorithm begins by using the fact that all K_4 's in Q are good. Thus for each $K \in Q$, there exists a colored vertex v such that v has at least one and at most 3 neighbors in K . The

algorithm chooses such a vertex v for each K , and if there are several such vertices, it selects one with a maximum number of neighbors in K . **Problem 1** shows why this algorithm chooses the vertex with the most neighbors in K . We call the selected vertex v an initial vertex. It might happen that several K 's in Q have the same initial vertex. Next the algorithm chooses one vertex u in each $K \in Q$, where u is adjacent to the initial vertex v of K . We refer to this chosen vertex u as K 's representative. Then the algorithm colors u with the color of v and removes the color from v . However, the above simple recoloring of a graph G can cause a recoloring problem which we call **Problem 2**. We discuss below a solution of Problem 2. Now we have a new coloring of a graph G , but we can't guarantee that it is a maximal coloring of the graph G . Therefore, we need to apply the procedure Extend on various sets of vertices, $V(Q)$, $V(T)$, R , and the set N of newly uncolored vertices, but doing this in the wrong order can also produce a recoloring problem which we call **Problem 3**. A solution to Problem 3 is also discussed below.

Problem 1. See Figure 5.2 and assume that an initial vertex of $K^1 \in Q$ is v_1 which has only one neighbor in K^1 , and v_1 also has three neighbors in $K^2 \in Q$ of which v_2 is an initial vertex. Suppose v_2 has only one neighbor in K^2 which is not adjacent to v_1 , and v_2 also has three neighbors in $K^3 \in Q$. Furthermore suppose v_3 is an initial vertex of K^3 and has one neighbor in K^3 which is adjacent neither to v_1 nor to v_2 . We also assume that the colors of v_1 , v_2 , and v_3 are the same. Then the number of uncolored K_4 's is reduced by the algorithm's recoloring. In this example as shown in Figure 5.2.(a), we have reduced the number of uncolored K_4 's by a factor of $1/3$. However, we might have string of n K_4 's in which case the number of uncolored K_4 's would have only reduced by a factor of $1/n < 1/4$. The algorithm avoids this problem by choosing, among all possible initial

(a) Without any restriction of choosing an initial vertex, recoloring may create unexpected new K_4 's.

(b) By choosing an initial vertex which is adjacent to a maximum number of vertices of a K_4 , the problem of (a) can be solved.

Figure 5.2. An example of Problem 1. All solid vertices have the same color.

vertices for K , one which is adjacent to a maximum number of vertices of K . Thus as shown in figure 5.2.(b), the algorithm actually uses v_1 as the initial vertex of K^1 and K^2 , and v_2 as the initial vertex of K^3 . Then the three K_4 's can be eliminated without creating any new ones.

(a) Given partially colored graph

(b) Recolored graph containing new K_4

Figure 5.3. An example of Problem 2. All solid vertices have the same color.

Problem 2. See Figure 5.3 and suppose that an uncolored vertex $u_0 \in U$ is a representative of $K \in Q$ and the initial vertex v_0 of K , having one neighbor in K , is adjacent to all three vertices in $t_0 \in T_i$, where i is the color of v_0 . And also suppose that there is no $t' \in T$, $t_0 \neq t'$, every vertex of which was adjacent to v_0 . Then after recoloring, $(K - u_0)$ may become a new element of T_i , and $(t_0 + v_0)$ may become a new uncolored K_4 . So the number of K_4 's in $G(U)$ would not decrease as we illustrate in Figure 5.3. To eliminate this problem, we need an additional step; When we remove the color from v_0 , choose any representative vertex z_0 of $t_0 \in T_i$ and color z_0 with i , and remove color i from the other vertex v_1 colored i and adjacent to z_0 . If v_1 is adjacent to all 3 vertices of $t_1 \in T_i$ and there is no $t' \in T_i$, $t_1 \neq t'$, every vertex of which was adjacent to v_1 , then uncolor v_1 , choose any representative vertex z_1 of t_1 , and color z_1 with i . Then continue

with v_j , t_j , and z_j , $j = 1, 2, \dots$, as long as possible. Figure 5.4 shows an example of this step. In the above iteration, a cycle can exist as shown in Figure 5.5, but the cycle does not generate a new K_4 - we show this in the next section. To implement this additional step in parallel, the algorithm introduces an auxiliary digraph H_i on $C_i \cup V(T_i)$, for $1 \leq i \leq \Delta-3$. The steps of building the digraph H_i are as follows:

- Step 1 : Start with no arcs. $\{ V(H_i) = C_i \cup V(T_i) \}$
- Step 2 : For each triangle $t \in T_i$, choose a representative vertex z .
- Step 3 : For all $v \in C_i$ and $t \in T_i$, add arc (v, z) if and only if v is adjacent to every vertex in t , and there is no $t' \in T$, $t' \neq t$, every vertex of which is adjacent to v .
- Step 4 : For each arc $(v, z) \in E(H_i)$, add an arc from z to the vertex $v' \in C_i$, where v' is different from v and adjacent to z .

(a) Given partially colored graph

(b) Recolored graph containing no K_4

Figure 5.4. An example of a method to solve Problem 2. All solid vertices have the same color.

(a) Given partially colored graph

(b) Recolored graph containing no K_4

Figure 5.5. An example of creating a cyclic iteration on recoloring. All solid vertices have the same color.

(a) Digraph of the example in Figure 5.4.

(b) Digraph of the example in Figure 5.5.

Figure 5.6. Examples of building digraphs.

Figure 5.6 (a) and (b) show a digraph H_i associated with the example in Figure 5.4 and Figure 5.5, respectively. We use the following notation: $v \in C_i$ is an end-vertex if and only if v 's outdegree in H_i is 0. Otherwise, we call v an inner-vertex. (These definitions will be used later.) Thus end-vertices, in Figure 5.6.(a), are v_2, w_2, w_4, w_6 , and w_8 . In Figure 5.6.(b), end-vertices are $w_2, w_4, w_6, w_8, w_{10}$, and w_{12} , i.e., any vertex of a component in H_i containing a cycle, is not an end-vertex. To recolor the graph G , the algorithm uses the digraphs H_i for $1 \leq i \leq \Delta-3$; colors z , the representative of $t \in T_i$, with color i , and removes color from $v \in C_i$ if and only if there exists a path in H_i from an initial vertex in C_i to z and v , respectively. For instance, in Figure 5.6.(a), vertices z_0 and z_1 are colored with i , and vertices v_0, v_1 , and v_2 are uncolored because there is a path from the initial vertex v_0 to each of those five vertices. As we can see, the result of this recoloring is exactly the same as part (b) of Figure 5.4. Similarly we shift colors along the cycle given in Figure 5.6.(b) and then achieve a recoloring of the graph in Figure 5.5

Problem 3. There are three problems that may occur when sequentially applying procedure Extend on each given set.

1. For each element in Q and T , where Q and T are the sets we have at the beginning of the Modify-Coloring Algorithm, the algorithm finds vertices not having a neighbor of every color and colors those vertices with their allowable colors. This coloring extension may cause a problem. Let us consider the following Figure 5.7. In Figure 5.7, v is an initial vertex of $K \in Q$, colored i , adjacent to every vertex of $t \in T_i$, and also adjacent to every vertex in $t' \in T$ or Q . If every vertex in t' has allowable colors different from one another, but no vertex in t has an allowable color, then we will color every vertex in t' with their allowable colors, and so this coloring extension may create another

(a) Given partially colored graph.

(b) Recolored graph containing new K_4

Figure 5.7. An example of Problem 3.1. All solid vertices are colored vertices and i, j, k , and l denote their colors.

uncolored K_4 because v 's color i has been shifted to u . To avoid this problem, we have to leave at least one vertex in t' uncolored, and color v with an allowable color. This can be accomplished by coloring only one vertex in t' . Then the vertex v is adjacent with at least 5 uncolored vertices and thus has an allowable color. The vertex v can then be colored by applying procedure Extend on N .

2. Applying the procedure Extend on the set R prior to doing it on the set N can cause another problem. Let us consider the example in Figure 5.8, where v was an initial vertex with respect to some K_4 (not shown) and v is adjacent to four vertices in R as

Figure 5.8. An example of Problem 3.2. A solid vertex has a color

shown. If after its color has been removed, the vertex v has only allowable color i and the vertex $w \in R$ adjacent to v has the same allowable color as v , then recoloring-problem can occur by coloring w with i . To avoid this problem, we should apply the procedure *Extend* on the set N before doing it on R .

3. Applying the procedure *Extend* simply on R can cause the third problem.

Consider the example in Figure 5.9, where v and u were initial vertices with respect to some K_4 's (not shown) and v and u are adjacent to the vertices in R as shown. Assume that both v and u originally had color i , and hence when we apply *Extend* to R , vertices x and y in R may have i as their only allowable color. Then coloring y with color i can also generate the recoloring-problem which can be solved by coloring x instead of y . Therefore, instead of applying *Extend* to all vertices in R , we apply *Extend* to such vertices x in R that are a part of a possible new uncolored K_4 , and have at least 4 uncolored neighbors including new uncolored vertices, such as v and u in Figure 5.9.

Figure 5.9. An example of Problem 3.3. All solid vertices have the same color.

Algorithm Modify Coloring

{ Input : a maximal $(\Delta-3)$ coloring of G without any bad K_4 's.

Output : a recoloring of G such that the number of K_4 's in $G(U')$ is at most $3/4$ times the number of K_4 's in $G(U)$.

The number of processors and the running time is dominated by step 44,57,and 61 which call the algorithm Extend or Luby's algorithm. Other steps use n^3 processors, P_{ijk} for $1 \leq i,j,k \leq n$.

Global memories used :

M is a connectivity matrix

Y is used to solve Problem 3.3

For $1 \leq c \leq \Delta-3$, H_c is a directed graph determined by color c ,

N_c indicates the set of newly uncolored vertices previously colored c .

For $1 \leq i \leq n$, $\text{String}[i]$ is a set of vertices in the same component of $G(U)$ as vertex i .

}

- 1: Let U be the set of uncolored vertices
- 2: Find a connectivity matrix M of $G(U)$ {refer to Chapter 3.}
- 3: for each processor P_{i11} , in parallel
 - { If vertex i is in an uncolored K_4 , then Q' will become a sorted string of the 4 vertices in that K_4 . }
- 4: $Q' = \{ \}$
 - { If vertex i is in an uncolored K_3 , then T' will become a sorted string of the 3 vertices in that K_3 . }
- 5: $T' = \{ \}$
 - { If uncolored vertex i is neither in Q' nor in T' , then a sorted string of the component containing vertex i will be assigned to R . If there is a new K_4 containing vertices of R after recoloring, RQ will become a string consisting of the vertices of that K_4 }
- 6: $R = \{ \}$ and $RQ = \{ \}$
 - { If vertex i is in its T' and each vertex of T' is adjacent to 2 vertices colored j , then T_s will be eventually equal to j . Otherwise, T_s will be 0.}
- 7: $T_s = 0$
 - { Determine the set which a vertex i belongs to }
- 8: for each processor P_{ij1} in parallel
- 9: if vertex i doesn't have a color and $M_{ij} = 1$
- 10: $\text{String}[i] = \text{String}[i] \cup \{j\}$

```

11:   for each processor  $P_{i11}$  in parallel
12:       if  $\text{String}[i]$  induces a  $K_4$ 
13:            $Q' = \text{String}[i]$ 
14:       else if  $\text{String}[i]$  induces a  $K_3$ 
15:            $T' = \text{String}[i]$ 
16:           if every vertex in  $T'$  has two neighbors colored  $c$ 
17:                $T_s = c$ 
18:       else  $R = \text{String}[i]$ 
        { Build a auxiliary digraph  $H_c$ , solving Problem 2 }
19:   for each processor  $P_{i11}$  in parallel
20:       Let  $c$  be either the color of vertex  $i$ , if vertex  $i$  has a color,
        or  $T_s$ , otherwise
21:        $H_c = H_c \cup \{i\}$ 
22:       if  $i$  is the first character in  $T'$  and  $T_s$  is not 0
23:           Let  $v_0$  and  $v_1$  be the vertices which are adjacent to  $i$  and colored  $c$ 
24:           for  $k = 0$  to 1
25:               if  $v_k$  does not have 3 neighbors in a  $K_3$  other than  $T'$ 
                { use  $n$  processors to do this comparison }
26:               add arc  $(v_k, i)$  in  $H_c$ 
27:               add arc  $(i, v_{1-k})$  in  $H_c$ 

```

28: for each processor P_{ij1} , in parallel
 29: if i is the first character in Q'
 { Choose an initial vertex v for each K in Q' and the representative u of K ,
 solving Problem 1.}
 30: Among the colored vertices having less than four neighbors in Q' ,
 choose a colored initial vertex v which has the maximum number
 of neighbors in Q' { Such v always exists because any such K is a
 good K_4 , and choosing such a v can be done in constant time,
 using n processors.}
 31: Let c be the color of v { thus v is a vertex in a digraph H_c }
 32: Choose a uncolored vertex u from v 's neighbors in Q'
 { note that $u \notin V(H_c)$ }
 33: Color u with c
 34: For each vertex w in H_c , mark w with c if there is a path vw in H_c
 { finding a path requires n processors and runs in $O(\log n)$ time. }
 { Recoloring some vertices of G }
 35: for each processor P_{i11} in parallel
 36: if i has been marked with c
 37: if vertex i has a color
 38: remove the color from i and put i into N_c
 39: else color i with c

```

    { Extend coloring to only one vertex in Q' or T', solving Problem 3.1 }
40:   for each processor  $P_{ijk}$ , in parallel
41:       if  $i$  is the first character of Q' or T'
42:           choose only one uncolored vertex  $v$  in Q' or T', if it exists, not
               having a neighbor of every color { this statement can be done in
               constant time, using  $n^2$  processors. }
43:           color  $v$  with an allowable color
44:   Extend  $(\cup N_i)$ 
       { Now we have a new set  $U$  of uncolored vertices }
45:   Find a connectivity matrix  $M$  of  $G(U)$ 
       { Find a component which contains a new uncolored  $K_4$  which comes from set  $R$ 
         and a newly uncolored vertex. }
46:   for each processor  $P_{ij1}$  in parallel
47:       if vertex  $i$  does not have a color
48:           if  $M_{ij} = 1$ 
49:                $String[i] = String[i] \cup \{j\}$ 
50:           if  $String[i]$  contains more than 4 elements
51:               if  $i$ 's neighbors in  $String[i]$  together with  $i$  form a  $K_4$ 
                   { requires  $n$  processors to do this comparison }
52:                $RQ =$  sorted string of vertices of the  $K_4$ 

```

```

    { Apply Extend to RQ, solving Problem 3.3}
53:   for each processor  $P_{i11}$ , in parallel
54:       if  $i$  is the first character in RQ
55:           choose a vertex  $v \in RQ$  with 4 or more uncolored neighbors in
                $G(U)$ .
56:            $Y = Y \cup \{v\}$ 
57:   Find a maximal independent set  $I$  of  $G(Y)$ 
58:   for each processor  $P_{i11}$ , in parallel
59:       if  $i \in I$ 
60:           color  $i$  with a color not occurring on its neighbors.
    { Apply Extend to all vertices of  $G$  }
61:   Extend ( $V(G)$ )

```

In order to show that Modify-Coloring Algorithm decreases the number of uncolored K_4 's to at most $3/4$ of its previous value, we first note the following facts and show necessary lemmas.

Fact 1. As we defined before, let U be the set of uncolored vertices at the start of Modify-Coloring Algorithm. Then any vertex in a $t \in T$ is not adjacent to vertices in $U - t$.

Fact 2. Just after line 39, every newly uncolored vertex is either an end-vertex or an inner-vertex.

In Modify-Coloring Algorithm, colors are represented by c . So we now use T_c as the set of uncolored components K_3 such that each vertex in the K_3 has two neighbors colored c .

Fact 3. Just after line 43, every inner-vertex originally colored c is adjacent to at least one uncolored vertex which was originally in some $t \in T_c$.

Fact 4. Just after line 44, every newly uncolored vertex in a new uncolored K_4 has exactly $\Delta-3$ colored neighbors.

Lemma 5.3. The number of newly uncolored end-vertices, right after executing line 39, is not greater than the number of K_4 's at the beginning of this algorithm

Proof. For each initial vertex v in a digraph H_i , there is at most one vw path, where w is an end-vertex, by the method of building a digraph H_i . Each K_4 has exactly one initial vertex, but each initial vertex may be an initial vertex for more than one K_4 . Therefore, the lemma is true. ■

Lemma 5.4. Let K be an uncolored K_4 which exists just after executing line 44 of Modify-Coloring Algorithm. Then K contains either two end-vertices, or one end-vertex v and $K-\{v\}$ was in R .

Proof. An uncolored K , just after line 44, must fall into at least one of the following classes.

- (1) K contains exactly one newly uncolored vertex z , and z is an inner-vertex.
- (2) K contains at least two inner-vertices.
- (3) K contains exactly one end-vertex and exactly one inner-vertex

(4) K contains exactly one newly uncolored vertex z and z is an end-vertex, and the remaining vertices are in $t \in T$.

(5) K contains exactly one newly uncolored vertex z and z is an end-vertex, and the remaining vertices are in $K \in Q$.

(6) K contains exactly one newly uncolored vertex z and z is an end-vertex, and the remaining vertices are in R .

(7) K contains at least two end-vertices.

Now we show that (1) - (5) cannot occur.

Case (1) cannot occur. If z is the only newly uncolored vertex in an uncolored K , then z 's uncolored neighbors span a K_3 . We know that the inner-vertex z is adjacent not only to an uncolored vertex v which was in a $t \in T_c$ by Fact 3, but also to a colored vertex which was also in t by the algorithm. However by Fact 1, the vertex v would not be in an uncolored K_3 which contains three originally uncolored vertices. Therefore, either z can be colored, or z 's uncolored neighbors cannot span a K_3 .

Case (2) cannot occur. Let vertices u and v be the inner-vertices in K originally colored c and c' , respectively. And also let x and y be the other vertices of K . Then by Fact 3, we may say that x and y are vertices of $t \in T_c$ and $t' \in T_{c'}$, respectively. But by Fact 1, since x is adjacent to y , $t = t'$. Hence $c = c'$ which implies u and v are not adjacent, which is a contradiction.

Case (3) cannot occur. To show this, we assume that vertex u is the lone end-vertex in K , and vertex v is the lone inner-vertex in K . Vertex u originally had color c , and vertex v had color c' . Since u and v are in K , we have edge $\{u, v\}$ and so c is not equal to c' . By Fact 2 the remaining two vertices in K , say x and y , are originally uncolored vertices, by Fact 3 v has an uncolored neighbor which was in some $t \in T_{c'}$ (we may say x is the uncolored neighbor.), and by Fact 1 y was also in t in order that x and y are adjacent

to each other. By the definition of T_c , x and y had exactly one neighbor colored c , namely u . Since u is now uncolored, vertices x and y have an allowable color c .

Case (4) cannot occur. In this case, an end-vertex v is the only newly uncolored vertex in K , and v 's uncolored neighbors span some $t \in T$. Let c be the original color of v . Then if $t \notin T_c$, one of three vertices in t would have been colored with c because they have an allowable color c . If $t \in T_c$, vertex v should be adjacent to every vertex of some $t' \in T$ in order that v became an end-vertex. In this case, v has more than three uncolored neighbors, i.e., v has an allowable color.

Case (5) cannot occur. Assume that an end-vertex v is the only newly uncolored vertex in K , and originally had color c . If the other three vertices were originally in $K' \in Q$ whose initial vertex was colored c' , $c \neq c'$, then these three vertices have an allowable color c , which is a contradiction. If $c = c'$, the initial vertex of K' has only one neighbor in K' while v has three neighbors in K' . This cannot occur because the algorithm chooses an initial vertex which has the maximum number of neighbors in K' among the colored vertices having less than four neighbors in K' . ■

Lemma 5.5. Let $G=(V,E)$ be a graph, where $|V| = n$ and the maximum degree of G is Δ . And also let I be a maximal independent set of G . Then $|I| \geq n/(\Delta+1)$.

Proof. Since I is a maximal independent set of G and Δ is the maximum degree of G , the maximum possible size of $V(G)$ is $|I| \times (\Delta+1)$ which occurs when every vertex in I has degree Δ in G and G has $|I|$ components. Therefore $n \leq |I| \times (\Delta+1)$, or $|I| \geq n/(\Delta+1)$. ■

Lemma 5.6. By executing lines 53 ~ 60 of Modify-Coloring Algorithm, the number of uncolored K_4 's containing only one end-vertex is decreased by a factor of at least $1/4$.

Proof. Let K be the newly uncolored K_4 , which contains one end-vertex v where the vertices of $K - \{v\}$ were in R . The algorithm chooses a vertex u in $K - \{v\}$ having a maximum degree in $G(U)$ and puts y into a set Y . Then $Y \subset R$, and $|Y|$ is the number of such K_4 's. Since any element y in Y has degree at least 4 in $G(U)$, y had an allowable color. The algorithm finds a maximal independent set I of $G(Y)$ and colors each vertex in the set I with its allowable color. Since coloring one vertex in Y means eliminating one newly uncolored K_4 , the number of elements in the set I determines the number of K_4 's eliminated. Now we note that the maximum degree of $G(R)$ is at most 3, and so that the maximum degree of $G(Y)$ is also at most 3. By lemma 5.5, therefore, the number of elements in the set I is at least $(1/4)$ times $|Y|$. The desired conclusion follows. ■

Theorem 5.7. The procedure Modify-Coloring decreases the number of uncolored K_4 's to at most $3/4$ of its previous value.

Proof. Let K be a newly uncolored K_4 . By lemma 5.4, K contains either two end-vertices, or one end-vertex and $K - \{v\}$ is in R . If K contains two end-vertices, we may say that by the algorithm, at least two K_4 's have been eliminated in order to generate K . If the newly uncolored K_4 's contain one end-vertex and $K - \{v\}$ was in R , by lemma 5.6 the number of such K_4 's has been decreased to $3/4$ of its previous value. Therefore, the worst case occurs when every newly uncolored K contains one end-vertex and $K - \{v\}$ is in R , and the result of that case is that the number of K_4 's is decreased to $3/4$ of its previous value. ■

5.3.5 Hajnal-Szemerédi Algorithm

Using the previous result, we can color a graph G with its maximum degree Δ colors. The main step is that first find a partial coloring with $\Delta-3$ colors such that the uncolored part has maximum degree 3 and does not contain any K_4 's, and then apply Karloff's algorithm.

Algorithm Hajnal-Szemerédi

{ Input : a graph G represented by an adjacency matrix, where maximum degree Δ of G is at least 4 and G does not contain any complete graph of $\Delta+1$ vertices or Δ is 3 and G contains K_4 's.

Output : Δ coloring of G

}

- 1: Run algorithm Extend on G
- 2: Let U be the set of uncolored vertices in G
- 3: while $G(U)$ contains a K_4
- 4: Run algorithm Eliminate
- 5: Run algorithm Modify-Coloring
- 6: Run Karloff's algorithm

Theorem 5.8. The algorithm Hajnal-Szemerédi runs in $O(\lg^3 n)$, using n^8 processors.

Proof. By Theorem 5.7, the "while" loop in Hajnal-Szemerédi algorithm iterates $O(\lg n)$ times, and all the procedures called in this algorithm runs in $O(\lg^2 n)$, using n^8 processors. Therefore, the running time of this algorithm is in $O(\lg^3 n)$ and the number of processors used is n^8 . ■

5.4 Closing Remarks

We have shown that there exist work-efficient parallel algorithms for Minimum Spanning Tree and Shortest Path problems, which are similar to their sequential algorithms and run in $O(n^2 / \lg n)$, using $O(\lg n)$ processors. However, no one has found work-efficient parallel algorithms for $\Delta+1$ Coloring and Δ Coloring problems. In part, this is due to the fact that their sequential algorithms cannot be easily parallelized or at least it is difficult to find a parallel version of their known sequential algorithms.

In parallelism, one goal is to find work efficient algorithms. Such algorithms can be considered optimal in a certain sense. However, in a different point of view, we would expect to have fast parallel algorithm for a given problem using any polynomial number of processors. Therefore, the class NC has been introduced, and some of the problems which have polynomial time sequential algorithms belong to this class and are said to have efficient parallel algorithms.

We finally note that as shown in this paper, $\Delta+1$ Coloring and Δ Coloring problems belong to the class NC and their efficient parallel algorithms are not easy.

REFERENCES

- Akl, Selim G. (1989), "The Design and Analysis of Parallel Algorithms," Prentice Hall, pp.251-277
- Baase, Sara (1988), "Computer Algorithms," Addison-Wesley, pp.145-171 and 361-396
- Berge, C. (1973), "Graphs and Hypergraphs," North Holland, Amsterdam.
- Cormen, Leiserson, and Rivest (1990), "Introduction to Algorithms," McGraw-Hill, pp. 688-729
- Dijkstra, Edsger W. (1959), "A note on two problems in connexion with graphs," Numerische Mathematik, vol.1, pp.269-271.
- Gibbons, Alan and Rytter, Wojciech (1988), "Efficient Parallel Algorithms," Cambridge, pp.1-87
- Goldberg, Andrew V. and Plotkin, Serge A. (1987), "Parallel $(\Delta+1)$ coloring of constant degree graphs," Information Processing Letters, 25(4), pp.241-245
- Hajnal, Peter and Szemerédi, Endre (1990), "Brooks Coloring in Parallel," SIAM J. DISC. MATH., vol. 3, no.1. pp.74-80
- Hakimi, S.L. (unpublished), "Proof of Brook's Theorem."
- Karchmer, Mauricio and Naor, Joseph (1988), "A Fast Parallel Algorithm to Color a Graph with Δ Colors," Journal of Algorithms 9, pp.83-91
- Karloff, Howard J. (1989), "An NC Algorithm for Brooks' Theorem," Theoretical Computer Science 68, pp.89-103
- Karloff, Howard J. (1985), "Fast Parallel Algorithms for Graph-Theoretic Problems," Ph.D. Thesis, Berkeley
- Luby, Michael (1985), "A simple parallel algorithm for the maximal independent set problem," Proc.17th Ann. ACM Symp. on Theory of Computing, pp.1-10
- Prim, R.C. (1957), "Shortest connection networks and some generalizations," Bell System Technical Journal, vol.36, pp.1389-1401.