Designing Efficient Parallel Algorithms for Graph Problems

Weifa Liang

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

September 1997
© Weifa Liang

Typeset in Computer Modern by \TeX and \LaTeX\textsuperscript{2e}.
Some of the results in Chapter 2 are joint work with Richard Brent and Hong Shen. These results have also appeared in [114, 121]. Chapter 3 is joint work with Brendan McKay and Hong Shen. The results of this chapter have also appeared in [120]. Chapter 4 is joint work with Brendan McKay. The content of this chapter has also appeared in [116]. Chapter 5 is joint work with George Havas and Brendan McKay. The results of this chapter have appeared in [115, 119]. Some of the results in Chapter 6 are the joint work with Xiaojun Shen [122]. These results are included in Sections 6.2, 6.3, and 6.5. Chapter 8 is joint work with Richard Brent. The contents of this chapter have also appeared in [113]. Some of Chapter 9 was carried out with Xiaojun Shen and Qing Hu. They are included in Sections 9.3 and 9.4. The results of this chapter have also appeared in [112, 123]. Chapter 10 is joint work with Brendan McKay, the contents of this chapter have also appeared in [118].

Except where otherwise indicated, this thesis is my own original work.

Weifa Liang
19 September 1997
Acknowledgments

I am truly indebted to my supervisors, Brendan McKay and Richard Brent, for their support and encouragement over the past three years. As a supervisor, Brendan made himself frequently available day and night to help me through research crises, clarify my ideas and explanations, and provide advice on research. He not only gave lots of his time and comments which influenced my writing but also showed me how to strive for a good presentation. His approach to algorithms, with the emphasis on identifying the combinatorial structure in the problem, the technique used and the search for simple and elegant solutions, has left a deep impact on me. I am sincerely thankful to him for being so generous with his time and for being so immensely patient with me during the course of my Ph.D. Richard was always available when I needed him. I enjoyed talking with him which always left me with some new ideas. I have learnt much from him about exploring a new research area. In addition to his technical expertise in many diverse areas, I have been particularly impressed by his scientific rigor and integrity, and I will strive to emulate these qualities in my own research. Both of my supervisors have had a great influence on my development as a researcher, their knowledge and intuition have been invaluable, and I am grateful to have been their student.

I would like to express my gratitude to Xiaojun Shen in The University of Missouri at Kansas City. It was a rare pleasure working with him when I was a visitor there. I have been benefited greatly from his insight, enthusiasm, and encouragement. I would also like to thank E. V. Krishnamurthy for his many suggestions and helpful comments in the preparation of this thesis.

I am grateful to my co-authors Richard Brent, George Havas, Qing Hu, Brendan McKay, Hong Shen, Xiaojun Shen, and Bingbing Zhou for their contributions to our collaborative work. Each has taught me a great deal about research and about writing about research.

Many other people have helped me in many different ways during my years in the Department of Computer Science at The Australian National University. They have helped to make my life in Canberra much easier and enjoyable. In particular, I would like to thank my colleagues Steve Blackburn who helped me with many of my \LaTeX{} questions over years, Bill Keating for his patient reading of my papers and providing his comments, Hongxue Wang for showing me lots of funny software. Many thanks are also due to our department head Brian Molinari for his generous financial sponsorship which has allowed me to attend several important international conferences, our system consultants Huge and Williamson for their help on my software problems, and our secretary Kathy for her administrative help.

Most of all, I especially want to thank my family. My parents for instilling in me the values I hold so dear and for giving me the confidence and drive to achieve my
goals. My wife and daughter for their love and patience that took me through many times of doubt and worry. Without their support and encouragement, this thesis could not possibly have been finished. I dedicate this thesis to them.
Abstract

Graph algorithms are concerned with the algorithmic aspects of solving graph problems. The problems are motivated from and have application to diverse areas of computer science, engineering and other disciplines. Problems arising from these areas of application are good candidates for parallelization since they often have both intense computational needs and stringent response time requirements. Motivated by these concerns, this thesis investigates parallel algorithms for these kinds of graph problems that have at least one of the following properties: the problems involve some type of dynamic updates; the sparsification technique is applicable; or the problems are closely related to communications network issues. The models of parallel computation used in our studies are the Parallel Random Access Machine (PRAM) model and the practical interconnection network models such as meshes and hypercubes.

Consider a communications network which can be represented by a graph $G = (V,E)$, where $V$ is a set of sites (processors), and $E$ is a set of links which are used to connect the sites (processors). In some cases, we also assign weights and/or directions to the edges in $E$. Associated with this network, there are many problems such as (i) whether the network is $k$-edge ($k$-vertex) connected with fixed $k$; (ii) whether there are $k$-edge ($k$-vertex) disjoint paths between $u$ and $v$ for a pair of given vertices $u$ and $v$ after the network is dynamically updated by adding and/or deleting an edge etc; (iii) whether the sites in the network can communicate with each other when some sites and links fail; (iv) identifying the first $k$ edges in the network whose deletion will result in the maximum increase in the routing cost in the resulting network for fixed $k$; (v) how to augment the network at optimal cost with a given feasible set of weighted edges such that the augmented network is $k$-edge ($k$-vertex) connected; (vi) how to route messages through the network efficiently. In this thesis we answer the problems mentioned above by presenting efficient parallel algorithms to solve them. As far as we know, most of the proposed algorithms are the first ones in the parallel setting.

Even though most of the problems concerned in this thesis are related to communications networks, we also study the classic edge-coloring problem. The outstanding difficulty to solve this problem in parallel is that we do not yet know whether or not it is in NC. In this thesis we present an improved parallel algorithm for the problem which needs $O(\Delta^{1.5} \log^3 \Delta \log n + \Delta^3 \log^4 n)$ time using $O(n^2 \Delta + n \Delta^3)$ processors, where $n$ is the number of vertices and $\Delta$ is the maximum vertex degree. Compared with a previously known result on the same model, we improved by an $O(\Delta^{1.5})$ factor in time. The non-trivial part is to reduce this problem to the edge-coloring update problem. We also generalize this problem to the approximate edge-coloring problem by giving a faster parallel algorithm for the latter case.

Throughout the design and analysis of parallel graph algorithms, we also find a
technique called the sparsification technique is very powerful in the design of efficient sequential and parallel algorithms on dense undirected graphs. We believe that this technique may be useful in its own right for guiding the design of efficient sequential and parallel algorithms for problems in other areas as well as in graph theory.
Contents

Acknowledgments v
Abstract vii
List of Figures xiii
Abbreviations xv

1 Introduction 1
1.1 The Parallel Computational Models 2
1.2 Notations and Terminology 4
1.3 The Problems and Related Literature Review 7
1.3.1 Testing for k-connectivity for fixed k 8
1.3.2 The optimal k-connectivity augmentation problem 8
1.3.3 Fully dynamic maintenance of k-connectivity 9
1.3.4 The partially dynamic maintenance of the solution of all pairs shortest paths 11
1.3.5 The k MVE MST problem and related problems 12
1.3.6 The approximate t-spanner problem 14
1.3.7 The edge-coloring problem and related problems 15
1.3.8 The maximal interlocking set problem 16
1.4 Overview of the Thesis 16

2 NC Algorithms for the Fully Dynamic Maintenance of k-Connectivity with $k = 2, 3$ 21
2.1 Introduction 21
2.2 Preliminaries 24
2.2.1 Basic concepts 24
2.2.2 The sparse k-edge (k-vertex) certificate 24
2.2.3 Eppstein et al's sparsification technique 25
2.3 The Maintenance of 2ECCs on Connected Graphs 26
2.3.1 The data structures 26
2.3.2 An algorithm for queries 29
2.3.3 Inserting and deleting edges 33
2.4 The Maintenance of 2- and 3-ECCs on General Graphs 34
2.4.1 2ECCs on disconnected graphs 34
2.4.2 An improved algorithm for 2ECCs 35
2.4.3 An algorithm for 3ECCs 35
2.5 The Maintenance of 2- and 3-VCCs on General Graphs .......... 36
2.5.1 Finding a strong, sparse k-vertex certificate of G .......... 36
2.5.2 The maintenance of 2VCCs .......................... 37
2.5.3 The maintenance of 3VCCs .......................... 38
2.6 Conclusions ................................................. 38

3 NC Algorithms for the Partially Dynamic Maintenance of the Solution of All Pairs Shortest Paths .......... 39
3.1 Introduction ................................................ 39
3.2 An Algorithm for the All Pairs Shortest Paths Problem .......... 41
3.3 Algorithms for Other Problems .......................... 43
3.3.1 Finding all pairs longest paths in a DAG ................. 43
3.3.2 Topological sorting in a DAG .......................... 44
3.3.3 The dynamic transitive closure problem ................ 45
3.4 Conclusions ................................................. 46

4 NC Algorithms for Testing k-Connectivity of Directed and Undirected Graphs with Fixed k .......... 47
4.1 Introduction ................................................ 47
4.2 Directed graphs ......................................... 48
4.3 Undirected graphs ....................................... 51
4.4 NC reduction of edge connectivity to vertex connectivity ..... 52
4.5 Conclusion ................................................. 52

5 NC Approximation Algorithms for the Optimal k-Connectivity Augmentation Problem with Fixed k .......... 53
5.1 Introduction ................................................ 53
5.2 Preliminaries .............................................. 56
5.3 2-Edge Connectivity Augmentation ........................ 57
5.4 Biconnectivity Augmentation ................................ 60
5.5 k-Edge Connectivity Augmentation ........................ 65
5.5.1 k-edge connectivity augmentation on a (k-1)-edge connected graph ............................................. 65
5.5.2 k-edge connectivity augmentation on a general graph .. 72
5.6 Conclusions ................................................. 73

6 Finding k Most Vital Edges in Minimum Spanning Trees .......... 75
6.1 Introduction ................................................ 75
6.2 Finding k Most Vital Edges ................................ 77
6.3 NC Algorithms for Finding the Single Most Vital Edge ....... 80
6.4 Sequential and Parallel Algorithms for k = 2, 3 ............... 83
6.4.1 Finding the first two most vital edges .................. 83
6.4.2 Finding the first three most vital edges ................ 86
6.5 Sequential and Parallel Algorithms with Fixed k ≥ 4 ...... 90
6.5.1 A sequential algorithm and its efficient implementation .. 90
|
|---|
|6.5.2 A parallel implementation | 94|
|6.6 Conclusions | 96|
|7 Finding the Single Most Vital Edge on Mesh and Hypercube Processor Arrays | 99|
|7.1 Introduction | 99|
|7.2 The Computational Models | 100|
|7.2.1 The 2-D mesh array | 100|
|7.2.2 The hypercube array | 101|
|7.3 Algorithms on the 2-D Mesh Array | 101|
|7.3.1 Finding the single most vital edge \( w.r.t. \) the MST problem | 101|
|7.3.2 Finding the single most vital edge \( w.r.t. \) SPs | 107|
|7.4 Algorithms on the Hypercube Array | 109|
|7.5 Conclusions | 113|
|8 Constructing the Spanners of Graphs in Parallel | 115|
|8.1 Introduction | 115|
|8.2 Preliminaries | 116|
|8.3 Finding Approximate \( t \)-Spanners | 117|
|8.3.1 Unweighted Graphs | 117|
|8.3.2 Weighted Graphs | 120|
|8.4 NC Algorithms for Finding Moderate \( 2t \)-Spanners | 123|
|8.4.1 The algorithm for unweighted graphs | 123|
|8.4.2 The algorithm for weighted graphs | 126|
|8.5 Conclusions | 127|
|9 Parallel Algorithms for Edge-Coloring | 129|
|9.1 Introduction | 129|
|9.2 Concepts and Notations | 130|
|9.3 An Edge-Coloring Update Algorithm | 131|
|9.4 A New Edge-Coloring Algorithm | 133|
|9.4.1 A simple version of the Karloff-Sllmoys’ algorithm | 133|
|9.4.2 An improved edge-coloring algorithm | 136|
|9.5 An Approximate Edge-Coloring Algorithm | 140|
|9.5.1 Graph decomposition | 140|
|9.5.2 An algorithm for approximate edge-coloring | 142|
|9.6 Conclusions | 145|
|10 An NC Algorithm for Computing Maximal Interlocking Sets | 147|
|10.1 Introduction | 147|
|10.2 Preliminaries | 148|
|10.3 The Algorithm | 149|
|10.3.1 The algorithmic description | 149|
|10.3.2 The correctness proof | 151|
|10.4 Conclusion | 153|
11 Conclusions and Open Problems     155
Bibliography                     157
List of Figures

2.1 An illustration of the proof of Lemma 2.12. .......................... 32
5.1 An example. ................................................................. 67
9.1 Inverting the α/β-path of \( v_2 \) affects the free color of \( w_1 \) in a fan of \( v_1 \), and inverting the α/β-path of \( v_3 \) affects the free color of \( z(v_2) \) in a fan of \( v_2 \). 135
10.1 An illustration. ............................................................ 153
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANLV</td>
<td>All nearest larger value</td>
</tr>
<tr>
<td>ANSV</td>
<td>All nearest smaller value</td>
</tr>
<tr>
<td>CC</td>
<td>Connected component</td>
</tr>
<tr>
<td>CREW</td>
<td>Concurrent read and exclusive write</td>
</tr>
<tr>
<td>CRCW</td>
<td>Concurrent read and concurrent write</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed acyclic graph</td>
</tr>
<tr>
<td>EREW</td>
<td>Exclusive read and exclusive write</td>
</tr>
<tr>
<td>LCA($x, y$)</td>
<td>The lowest common ancestor of $x$ and $y$</td>
</tr>
<tr>
<td>MST</td>
<td>Minimum spanning tree</td>
</tr>
<tr>
<td>MSF</td>
<td>Minimum spanning forest</td>
</tr>
<tr>
<td>MWSC</td>
<td>Minimum weighted set cover</td>
</tr>
<tr>
<td>PRAM</td>
<td>Parallel random access machine</td>
</tr>
<tr>
<td>2ECC</td>
<td>2-Edge connected component</td>
</tr>
<tr>
<td>3ECC</td>
<td>3-Edge connected component</td>
</tr>
<tr>
<td>2VCC</td>
<td>2-Vertex connected component</td>
</tr>
<tr>
<td>3VCC</td>
<td>3-Vertex connected component (triconnected component)</td>
</tr>
<tr>
<td>$k$ MVE</td>
<td>The $k$ most vital edges</td>
</tr>
<tr>
<td>SP</td>
<td>Shortest path</td>
</tr>
</tbody>
</table>
LIST OF FIGURES
Chapter 1

Introduction

As more computers have incorporated some form of parallelism, the emphasis in algorithm design has shifted from sequential algorithms to parallel algorithms, i.e., algorithms in which multiple operations are performed simultaneously. Consequently, the study of parallel algorithms has become a key part of parallel processing because it improves our understanding of parallelism. As an important branch of parallel algorithms, the study of parallel graph algorithms has been of growing interest in the last decade. Since we already knew that many problems in computer science and other disciplines can be modeled as graph problems, the development of efficient sequential/parallel algorithms to solve these graph problems is very important. As result of much effort in the past decades, there exist many efficient sequential graph algorithms [2] and corresponding design techniques such as depth-first search (DFS), etc. However, most of these algorithms are inherently sequential, and the techniques involved such as DFS are hard to parallelize [150]. In order to solve these graph problems in the parallel environment, people have developed some efficient parallel algorithms. Meanwhile, in order to design such parallel algorithms, efficient parallel design techniques such as Eulerian traversal, tree contraction etc. have been introduced. An extensive survey of parallel algorithms and techniques for graph problems can be found in [94, 98, 110, 144, 170]. It must be mentioned that most graph problems studied in the parallel context until now are fundamental problems such as connected components, minimum spanning trees, maximal independent set and so on, whereas some complicated problems such as the problems arising from communications networks are rarely studied in this environment. Furthermore, the study in the past focused on static versions, which means there is no change of the input during the computation. Unfortunately, the real world always changes dynamically. So, how to design parallel algorithms for graph problems that undergo some kind of change, and what kind of technique can be applicable to improve the performance of parallel algorithms are new topics. Motivated by these concerns, in this thesis we will investigate parallel solutions to a class of graph problems that share at least one of the following properties: the problems involved have some type of dynamic updates; the sparsification technique (see below) is applicable; or the problems are related to reliability issues of communications networks.

Before we proceed, we give a brief explanation of dynamic updates, the sparsification technique and the reliability issue of communications networks. Let $G = (V, E)$ be a directed/undirected graph where $V$ is the vertex set and $E$ is the edge set. Assume
that $|V| = n$ and $|E| = m$. Let $\mathcal{P}$ be a property of $G$ such as a minimum spanning tree, an edge coloring, and so on. The dynamic update usually means, re-computing the property $\mathcal{P}$ on the resulting graph after some changes are applied on $G$ such as adding an edge ($k$ edges) to and/or deleting an existing edge ($k$ edges) from $G$, or adding a new vertex and all edges incident to this vertex to $G$. It is expensive to run the algorithm for finding $\mathcal{P}$ on this resulting graph from scratch. A better way to deal with the case of minor updates is to build some data structures initially, and to maintain these data structures efficiently when an update occurs. The aim is to make a query of the property $\mathcal{P}$ in the resulting graph easy to answer.

The sparsification technique can be described as follows. When computing a property $\mathcal{P}$ of $G$, instead of finding this property from $G$, we first find a sparse subgraph $G'$ (or in general an auxiliary graph that is not necessarily a subgraph of $G$ at all) such that the $\mathcal{P}$ of $G'$ is identical to the $\mathcal{P}$ of $G$. Then we compute the property $\mathcal{P}$ of $G'$. This gives the property $\mathcal{P}$ of $G$. Let $T(n, m)$ be the time needed to find $\mathcal{P}$ from $G$ and $T_1(n, m)$ be the time to build $G'$ from $G$. Obviously, the sparsification technique is useful if and only if $T_1(n, m) + T(n', m') < T(n, m)$ where $n'$ is the number of vertices and $m'$ is the number of edges in $G'$. It should be mentioned that different $G'$ need to be found for different $\mathcal{P}$.

Let us consider a computer communications network which can be modeled as a graph $G(V, E)$, where $V$ is a set of sites (processors), and $E$ is a set of links which are used to connect the sites (processors). In some cases, we also assign weights and/or directions to the edges in $E$. The reliability issue of the network involves whether the network is still connected and the sites in the network can communicate with each other when some sites and/or links fail; and how to route messages through the network efficiently.

The rest of this chapter is organized as follows. We begin by introducing the models of parallel computation which will be used for developing our parallel algorithms. Next we introduce some notations and terminology for later use. Third, we define the problems that will be investigated in this thesis. Meanwhile, we give an extensive review of related work to these problems. Finally we conclude this chapter by presenting our results and showing the structure of the thesis.

1.1 The Parallel Computational Models

A general goal in the design of parallel algorithms is to find algorithms which solve the problem faster than on a uniprocessor machine, while using as many processors as necessary. Let $\mathcal{A}$ be a parallel algorithm to solve a problem of size $n$ in time $t(n)$ using $p(n)$ processors, then the amount of work of $\mathcal{A}$ can be expressed as $w(n) = p(n) \times t(n)$. If $w(n)$ is within a constant factor of the time complexity of the best sequential algorithm for the problem, then $\mathcal{A}$ is said to achieve linear speedup, and if $w(n)$ is within a constant factor of the sequential lower bound for the problem, then $\mathcal{A}$ is said to achieve optimal speedup. If $w(n) = T(n) \times \log^c n$ where $T(n)$ is the time complexity of the best sequential algorithm for the problem and $c$ is a constant, then $\mathcal{A}$ is said to be efficient.

The parallel computational model can be classified into two general classes. One
is the *shared memory model* which is further divided into three subclasses depending on the restrictions imposed on read and write. Another is the *interconnection network model* in which communication cost is the major concern, and the computational complexity depends on the interconnection topology structure of interconnection networks.

When considering a problem that is not well understood in a parallel environment, it is often a good strategy to begin with a shared memory model. In this way, the inherent parallelism of the problem can be explored without paying much attention to the interconnection topology structure of the network and the other factors. After a problem is well studied in the shared memory model, it can be simulated step by step in a real parallel computer using its interconnection structure. It must be mentioned that the parallel algorithm obtained through the simulation is usually not optimal. To explain this phenomenon is not hard because a specific interconnection topology structure may have some unique properties and the simulation may not fully utilize these properties.

In this thesis we will use both the Parallel Random Access Machine model, or PRAM, and the interconnection network model. The PRAM is a shared memory machine in which many processors may have access to a single shared memory unit. More precisely, the shared memory model consists of a number of processors, each of which has its own local memory and executes its own local program, and which communicate by exchanging data through a shared memory unit. Each processor is uniquely identified by an index, called a *processor number* or *processor id*, which is available locally. In the PRAM model there is the possibility of read- and write-conflicts, in which two more processors try to read from or write into a shared memory unit concurrently. The distinctions in the way these conflicts are handled lead to several different variants of this model. (i) The *weakest* of these is the Exclusive Read and Exclusive Write (EREW) PRAM in which concurrent read and concurrent write are forbidden; (ii) intermediate is the CREW PRAM in which Concurrent Read is allowed but only Exclusive Write is permitted when write conflicts happen; (iii) *strongest* is the CRCW PRAM in which both Concurrent Read and Concurrent Write are allowed. Several varieties of CRCW PRAMs have been defined; they differ in the method of resolving the write conflicts. One of the least restrictive methods used to resolve the write conflict is the ARBITRARY model in which any processor participating in a concurrent write may succeed, and the algorithm always work correctly regardless of which one succeeds. The other versions of the CRCW PRAM are the COMMON and the PRIORITY models. The COMMON model requires that all processors writing to a shared memory unit must write the same value, and in the PRIORITY model there is a natural linear ordering on the processors participating in the write, and only the write of the processor with the highest priority right succeeds. A detailed description of the PRAM model can be found in [94, 98].

Even though there is a variety of PRAM models, they do not differ very widely in their computational power. Some studies have been done about the simulations between different PRAMs. It is shown that any algorithm for a PRIORITY CRCW PRAM can be simulated by an EREW PRAM with the same number of processors and with the parallel time increased by only a factor of $O(\log r)$, where $r$ is the number
of processors [43, 169]. This can be done as follows. Let \( P_1, P_2, \ldots, P_r \) be the processors and \( M_1, M_2, \ldots, M_r \) be the memory locations used by the CRCW PRIORITY algorithm. The simulating EREW PRAM algorithm uses \( r \) auxiliary memory locations \( N_1, N_2, \ldots, N_r \) for simulating a write or read step. If processor \( P_i \) needs to access location \( M_j \) in the CRCW PRIORITY algorithm, then it writes the ordered pair \((j, i)\) in location \( N_j \). The array \( N_j, j = 1 \ldots, r \) is then sorted in lexicographically increasing order in \( O(\log r) \) time using the \( r \) processors. Then, by reading adjacent entries in this sorted array, the highest priority processor accessing any given location can be determined in constant time. For a write instruction, these processors then execute the write as specified in the CRCW PRIORITY algorithm. For a read instruction, the processors read the specified locations, and then in additional \( O(\log r) \) time, duplicate the value read so that there are enough copies of each value for all the processors that need to read it.

Further, any algorithm for a PRIORITY CRCW PRAM can be simulated by a COMMON CRCW PRAM with no loss in parallel time provided sufficiently many processors are available [106]. Let \( P_1, P_2, \ldots, P_r \) be the processors used by the CRCW PRIORITY algorithm. The simulating CRCW COMMON algorithm uses auxiliary processors \( P_{i,j} \) and memory locations \( M_j, 1 \leq i, j \leq r \). The locations \( M_j \) are initialized to 0. Processor \( P_{i,j}, i < j \), determines the memory addresses \( M_i \) and \( M_j \) that processors \( P_i \) and \( P_j \) were to access in the PRIORITY algorithm and writes a 1 in location \( M_j \) if \( M_j = M_i \). Now \( P_j \) can ascertain if it is the lowest-numbered processors that needs to write into \( M_j \) by testing if \( M_j \) is still 0. If so, it writes into \( M_j \) the value it was supposed to write by the CRCW PRIORITY algorithm.

The interconnection network model can be viewed as an undirected simple graph \( G = (V, E) \), where each vertex \( i \in V \) represents a processor, and each edge \((i, j) \in E\) represents a two-way communication link between processor \( i \) and processor \( j \). Assume that each processor has its own local memory, and no shared memory is available. The network models fully depend on the interconnection topology structures. A different interconnection topology structure corresponds to a different model. Some popular models are butterfly, mesh, mesh-of-tree, pyramid, hypercube, shuffle-exchange etc. The book of Leighton [110] is an excellent reference to the interconnection network models.

Note that the PRAM model does not consider the communication issue among processors. In order to overcome the drawback of the PRAM model in this aspect, some more realistic models such as BSP [168], Log P [34] and QRQW [73] have been proposed recently. Because of its simplicity, the PRAM model is assumed in all our discussions in the remainder of the thesis except in Chapter 7 unless specified.

1.2 Notations and Terminology

In this section we present some definitions, notations and terminology from graph theory which will be used throughout the thesis. A simple graph \( G = (V, E) \) is a finite vertex set \( V \) and a finite edge set \( E \), and there is an edge at most between a pair of vertices, where \(|V| = n\) and \(|E| = m\). If \( G \) is undirected, we use \((u, v)\) to represent an
undirected edge between \( u \) and \( v \). Otherwise, we use \( \langle u, v \rangle \) to represent a directed edge in which vertex \( u \) is the head and vertex \( v \) is the tail. A graph \( G \) is a bipartite graph if \( V \) can be partitioned into two non-empty sets \( X \) and \( Y \), so that each edge has one endpoint in \( X \) and one endpoint in \( Y \).

A subgraph \( G' = (V', E') \) of \( G \) is a graph such that \( V' \subseteq V \) and \( E' \subseteq E \). For an edge \( e = (u, v) \), we say that \( u \) and \( v \) are adjacent in \( G \), and \( u \) and \( v \) are the endpoints of \( e \). A weighted graph is a graph with a weight function on its edges, i.e., it is a graph together with a function \( W : E \to \mathcal{R} \). The degree of a vertex \( v \) in \( G \), denoted by \( d(v) \), is the cardinality of the set \( \{ u \mid (u, v) \in E \} \). Define \( \Delta = \max \{ d(v) \mid v \in V \} \) and \( \delta = \min \{ d(v) \mid v \in V \} \). The degree of \( v \) in a subgraph \( H \) of \( G \) is denoted by \( d_H(v) \). A graph \( G \) is an Eulerian graph if \( G \) does not contain any odd degree vertex.

Let \( U \) be a subset of vertices, the graph \( G_U = (U, E') \) induced on \( U \) is a subgraph of \( G \) such that \( U \subseteq V \) and \( E' = (U \times U) \cap E \). A graph \( G' = (V, E') \) is a spanning subgraph of \( G \) if \( E' \subseteq E \).

A path in \( G \) between \( v_1 \) and \( v_k \) is a vertex sequence \( v_1, v_2, \ldots, v_k \) such that \( (v_i, v_{i+1}) \in E \) for all \( i \), \( 1 \leq i < k \). We assume that each vertex in the sequence appears only once. Let \( P \) be a path between \( v_1 \) and \( v_k \), and \( (v_k, v_1) \) be an edge that does not appear on \( P \), then \( P \) and \( (v_k, v_1) \) forms a cycle. The length of a path is the weighted sum of the edges in the path. A shortest path between \( u \) and \( v \) is a path which has the minimum weighted sum of edges. The distance between \( u \) and \( v \) is the length of the shortest path between them. A graph \( G \) is connected if there exists a path between any two vertices in \( G \); \( G \) is disconnected otherwise. If \( G \) is disconnected, find all maximal connected subgraphs, each of the subgraphs is a connected component (CC for short) of \( G \).

A graph \( T \) is a tree if \( T \) is a connected graph that does not contain any cycle. A subgraph \( T \) of \( G \) is a spanning tree of \( G \) if \( T \) is a tree and \( T \) is a spanning subgraph of \( G \) as well. A tree \( T \) is a minimum spanning tree (MST for short) of \( G \) if \( T \) is a spanning tree of \( G \) with the minimum weighted sum of edges in \( T \). An inverted tree is a directed tree with the edges directed towards the root and there is a self-loop at the root.

A directed graph \( G \) is a directed acyclic graph (or DAG) if \( G \) does not contain any directed cycle.

A subgraph \( G' \) of \( G \) is a \( t \)-spanner (an approximate \( t \)-spanner) of \( G \) if, for every \( u \in V \) and \( v \in V \), the distance between \( u \) and \( v \) in \( G' \) is at most \( t \) \( f(t) \) times longer than the distance in \( G \) where \( f(t) \) is a polynomial function of variable \( t \) and \( t \leq f(t) < \infty \), and \( t \) \( f(t) \) is the factor of \( G' \). There are two criteria to measure the sparseness of a spanner, that is, the size, defined as the number of edges in the spanner, and the weight, defined as the weighted sum of the edges in the spanner. An MST of \( G \) is obviously the sparsest spanner in terms of both size and weight, but its factor can be as bad as \( n - 1 \) [141]. For convenience, we denote by \( wt(MST) \), the weighted sum of the edges in the MST. Usually the sparseness of a spanner is judged by comparing it to the size and the weight of the MST.

A branching of a directed graph rooted at \( r \) is a spanning tree of this graph such that each vertex except \( r \) has in-degree exactly one and \( r \) has in-degree zero. The weight of a branching is the weighted sum of the edges in the branching. An optimal branching is a branching with the minimum weight.
A minimum edge cut of a graph $G$ is a nontrivial partition $(A, \overline{A})$ of $V$ such that the total number of edges between $A$ and $\overline{A}$ is minimum where $\overline{A} = V - A$. We also use the name minimum edge cut for that set of edges, which should not cause confusion.

An independent set of $G$ is a vertex set in which no edge exists between any two vertices. A maximal independent set of $G$ is an independent set such that no more vertices can be added to it without introducing an edge.

A matching of $G$ is a set of edges such that any vertex of $G$ is incident to at most one edge in it. A vertex is called free if it is not incident to any edge in the matching. A maximal matching of $G$ is a matching that there is no any edge between any two free vertices.

A maximal path of $G$ is a simple path $P$ starting from a vertex $r$ such that $P$ cannot be extended without encountering a vertex that is already on $P$ [4].

A vertex $v \in V$ in $G$ is an articulation point if the deletion of $v$ and the edges incident to $v$ leaves $G$ disconnected. Similarly, an edge $e \in E$ in $G$ is a bridge if the deletion of $e$ leaves $G$ disconnected.

A vertex $v \in V$ dominates another vertex $u$ of $V$ if and only if $(u, v) \in E$, i.e., $u$ is an adjacent vertex of $v$. Assume that there are two vertex-disjoint sets $A$ and $B$ of $V$, we say $A$ dominates $B$ if, for every vertex $u \in B$, there exists at least a vertex $v \in A$ such that $u$ is dominated by $v$.

The k-connectivity of $G$ refers to either $k$-edge connectivity or $k$-vertex connectivity. A graph $G$ is $k$-edge connected if it is still connected after deleting any $k - 1$ edges from $G$. A graph $G$ with $|V| \geq k + 1$ vertices is $k$-vertex connected if it is still connected after deleting any $k - 1$ vertices from $G$. If a graph is not $k$-edge/$k$-vertex connected, it contains $k$-edge/$k$-vertex connected components (KECCs/kVCCs for short). Given $G$ and an integer $k$, a pair of vertices $u$ and $v$ is said to be $k$-edge connected if the removal of any $k - 1$ edges in $G$ leaves $u$ and $v$ connected. This is an equivalence relationship, and we write it by $\equiv_k$, i.e., if a pair of vertices $x$ and $y$ is $k$-edge connected we write $x \equiv_k y$. The vertices in $G$ are partitioned by this relationship into equivalence classes called $k$-edge connected components. It seems to be difficult to define $k$-vertex connected component with $k > 3$, here we only define this concept for $1 < k \leq 3$, which will be used in the thesis. Given $G = (V, E)$, let $E_1, E_2, \ldots, E_h$ be the partition of $E$ into equivalence classes such that two edges $e'$ and $e''$ are in the same equivalence class if and only if either (i) $e' = e''$ or (ii) there is a simple cycle of $G$ containing both $e'$ and $e''$. For all $i$, $1 \leq i \leq h$, Let $V_i$ be the set of vertices that are endpoints of the edges in $E_i$. The subgraph $G_i = (V_i, E_i)$ are called the 2-vertex connected component (2VCC for short). An edge contained in no cycle is a 2VCC by itself. Usually, 2-vertex connectivity is called biconnectivity, a 2-vertex connected component is called a biconnected component or a block. A 2-vertex connected graph is called biconnected graph. Let $\{a, b\}$ be a pair of vertices in a biconnected graph $G$. Suppose the edges of $G$ are divided into equivalence classes $E_1, E_2, \ldots, E_p$ such that two edges which lie on a common path not containing any vertex of $\{a, b\}$ except as an endpoint are in the same class. The classes $E_i$ are called the separation classes of $G$ with respect to $\{a, b\}$. If there are at least two separation classes, then $\{a, b\}$ is a separation pair of $G$ unless (i) there are exactly two separation classes, one class consists of a single
edge, or there are exactly three classes, each consisting of a single edge. If \( G \) is a biconnected graph such that no any separation pair in it, then \( G \) is triconnected. Let \( \{a, b\} \) be a separation pair of \( G \). Let the separation classes of \( G \) with respect to \( \{a, b\} \) be \( E_1, E_2, \ldots, E_p \). Let \( E' = \bigcup_{i=1}^k E_i \) and \( E'' = \bigcup_{i=k+1}^p E_i \) be such that \( |E'| \geq 2 \) and \( |E''| \geq 2 \). Let \( G_1 = (V(E'), E' \cup \{a, b\}) \), \( G_2 = (V(E''), E'' \cup \{a, b\}) \). The graphs \( G_1 \) and \( G_2 \) are called split graphs of \( G \) with respect to \( \{a, b\} \). Replacing \( G \) by two split graphs is called splitting \( G \). The new edges \( \{a, b\} \) added to \( G_1 \) and \( G_2 \) are called virtual edges. If \( G \) is biconnected, then any split graph of \( G \) is also biconnected. Suppose \( G \) is split, the split graphs are split, and so on, until no more splits are possible. The graphs constructed in this way are called split components of \( G \). The split components of \( G \) are of three types: triple bonds of form \( \{(a, b), (a, b), (a, b)\} \), triangles of the form \( \{(a, b, c), (a, b, c), (b, c)\} \), and triconnected graphs. Then the split components of \( G \) are called triconnected components of \( G \). Usually, a triconnected component is also called 3-vertex connected component (3VCC for short), and 3-vertex connectivity is called triconnectivity.

Let \( X = \{x_1, x_2, \ldots, x_n\} \) be a finite set, and let \( E = \{E_i \mid i \in I\} \) be a family of subsets of \( X \). The family \( E \) is said to be a hypergraph on \( X \) if (i) \( E_i \neq \emptyset \) for all \( i \in I \) and (ii) \( \bigcup_{i \in I} E_i = X \), where \( I \) is an integer set. The couple \( H = (X, E) \) is called a hypercure. The \( X \) is called the vertex set and the sets \( E_1, E_2, \ldots, E_m \) are called the edges. Given a hypergraph \( H = (V, E) \) and a weight function on the vertices \( w : V \to \mathbb{R} \), the minimum weight set cover (MWSC for short) problem consists of finding a minimum weighted subset \( V' \subseteq V \) that covers \( H \); i.e., an \( V' \) that minimizes \( w(V') = \sum_{v \in V'} w(v) \) subject to the constraint that \( \{u_i, v_i\} \cap V' \neq \emptyset \) for all \( v_i = (u_i, v_i) \in E \). This is equivalent to the problem of, given a set system \( \mathcal{A} \subseteq 2^X \) and a weight function \( w : \mathcal{A} \to \mathbb{R} \), finding a minimum weighted sub-collection \( \mathcal{A}' \subseteq \mathcal{A} \) such that \( \bigcup \mathcal{A}' = X \). The unweighted set cover problem is the special case when all the weight are 1. Since this problem is NP-complete [97], there is no any polynomial algorithm for it unless \( P = NP \).

In the design of sequential algorithms, it is well known that an algorithm can be classified either in \( P \) or \( NP \) [88], where \( P \) represents the algorithm run in polynomial time, and \( NP \) represents the algorithm not run in polynomial time. In the parallel environment there is a similar classification. A parallel algorithm can be classified into either NC or P-complete, where the class NC comprises problems that can be solved in time \( t(n) = O(\log^c n) \) using \( O(n^{O(1)}) \) processors and \( c \) is a constant, the class P-complete comprises those problems that can be solved in polynomial time but not in NC unless \( P = NC \). A good reference for this topic is [77].

A fully dynamic graph algorithm is one that allows edge/vertex insertions and deletions, and recomputes some desired graph property quickly after each update. A partially dynamic graph algorithm is one that allows either edge/vertex insertions or edge/vertex deletions but not both of them.

1.3 The Problems and Related Literature Review

Let us consider \( G \) as the communications network defined initially. Concerning this network, we can ask the following questions.
1.3.1 Testing for $k$-connectivity for fixed $k$

A simple question about the communications network is to ask whether it is $k$-vertex ($k$-edge) connected, i.e., whether there are $k$-vertex ($k$-edge) disjoint paths between any pair of vertices for fixed $k$. If the network is $k$-vertex ($k$-edge) connected, then when $k - 1$ processors ($k - 1$ links) fail, the communication network is still connected, and the all remaining processors in it can communicate with each other.

The $k$-connectivity of $G$ is a basic property in graph theory with wide applications in augmented compilers, distributed computings, reliable communications networks, etc. Extensive studies have been conducted in the past decades. Some efficient sequential algorithms for this problem have been found by Even [47] and Gabow [62]. Recently, incorporating the sparse $k$-edge/$k$-vertex certificate concept [135, 136], Eppstein et al. [44, 45] introduced a sparsification technique [44], they are able to improve the time complexity for the problem from $T(n, m)$ to $T(n, m/n)$ when $k$ is fixed, where $T(n, m)$ is the original time complexity of the algorithms of Even or Gabow. In the parallel context, it is still unknown to decide whether or not a graph is $k$-edge/$k$-vertex connected is in NC, but for an undirected graph $G$ with fixed $k$, Khuller et al. [101] have proven that testing for $k$-edge connectivity of $G$ is in NC, and Cheriyan et al. [21] have shown that testing for $k$-vertex connectivity of $G$ is in NC too. However, for a directed graph $G$ with fixed $k$, whether testing for $k$-edge/$k$-vertex connectivity is in NC was previously unknown. In this thesis we give an affirmative answer for this latter case by showing it is in NC.

1.3.2 The optimal $k$-connectivity augmentation problem

The robustness of a communications network subject to link failures is often modeled by the edge connectivity of the associated graph. Consequently there is interest in knowing how to increase the connectivity by adding further edges, with some cost measure minimized. This “connectivity augmentation problem” has received a good deal of attention during the past two decades [46, 52, 56, 84, 86, 87, 103, 138, 172, 173].

Generally speaking, given an undirected graph $G = (V, E_0)$ which is not $k$-edge ($k$-vertex) connected, and a feasible set $E$ of weighted edges on $V$ such that $G(V, E_0 \cup E)$ is $k$-edge ($k$-vertex) connected with $|E| = m$, the optimal $k$-edge ($k$-vertex) connectivity augmentation problem of $G$ is to find a subset $S^* \subseteq E$ such that $G(V, E_0 \cup S^*)$ is $k$-edge ($k$-vertex) connected and the weighted sum of the edges in $S^*$ is minimized.

If $G$ is an unweighted graph, and the edges in the feasible set $E = E(K_n) - E_0$ are unweighted where $E(K_n)$ is the edge set of the complete graph $K_n$ on the vertex set $V$, there are many algorithms for this problem which deliver exact solutions [46, 62, 138, 86, 87, 85, 152, 172, 173]. However, when the edges in the feasible set $E$ are weighted, the situation is totally different. It is well known that the optimal $k$-connectivity augmentation problem is NP-complete even for $k = 2$ [46, 56]. So, it is unlikely to solve this problem in polynomial time unless $P = NP$. Here we only consider polynomial approximation algorithms for it. When $k = 2$, Frederickson and JáJá [56], and Khuller and Thurimella [103] presented efficient sequential algorithms for this case by delivering a solution within twice the optimum if $G = (V, E_0)$ is connected.
A closely related problem is to find a minimum weighted/cardinality $k$-edge ($k$-vertex) connected subgraph in a $k$-edge ($k$-vertex) weighted/unweighted connected graph. By considering the case of $E_0 = \emptyset$ and the case where the edges in $E_0$ have zero weights, it is easy to see that this problem is equivalent to the optimal $k$-connectivity augmentation problem. Obviously it is also NP-complete. There were some approximation results for this latter problem in the past. When $G$ is unweighted and $k = 2$, Khuller and Vishkin [104] presented the first $O(m)$ time algorithms which deliver a solution for finding a minimum cardinality 2-edge connected subgraph within 1.5 times optimum, and a solution for finding a minimum cardinality biconnected subgraph within 5/3 times optimum. Later, by improving analysis of the Khuller and Vishkin algorithm, Garg et al. [69] showed the solutions by the algorithms of Khuller and Vishkin are respectively within 1.25 times optimum and 1.5 times optimum for these two problems. Recently Chong and Lam [25] developed NC approximation algorithms for the problems above using the approximating maximum cardinality matching technique [52]. The solution by their NC algorithm for finding a minimum cardinality 2-edge connected subgraph is $1.5 + \epsilon'$ times optimum, and the solution for finding a minimum cardinality biconnected subgraph is $7/4 + \epsilon'$ times optimum, where $\epsilon'$ is an arbitrarily positive constant. As for general $k$, Khuller and Raghavachari [102] recently obtained a solution for finding a minimum cardinality $k$-edge connected subgraph within 1.85 times optimum which is the best result so far. When $G$ is weighted, to find an approximation solution for the minimum weighted $k$-edge ($k$-vertex) connected subgraph problem seems more difficult. For $k = 2$ this special case, the approximation algorithm of Frederickson and JáJá for the optimal $2$-connectivity augmentation problem [56] implies a solution for this problem within 3 times optimum. Khuller and Vishkin [104] further improved the solution for finding a minimum weighted 2-edge connected subgraph, and presented a solution which is twice the optimum by applying the weighted matroid intersection algorithm of Gabow [62]. Very recently, Khuller and Raghavachari [102] obtained an approximation solution for finding a minimum biconnected subgraph within $2 + \frac{1}{k}$ times optimum by applying the submodular flow technique of Frank and Tardos [53]. Ravi and Williamson [149] presented an approximation solution for the problem of finding a minimum weighted $k$-vertex connected subgraph with arbitrary $k$. Their solution is $2\mathcal{H}_k$ times optimum, where $\mathcal{H}_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$.

It must noted that most of the algorithms mentioned above are inherently sequential, it seems too hard to parallelize them efficiently. We do not know of any NC approximation algorithm for the optimal $k$-connectivity augmentation problem when a feasible set of weighted edges is given with fixed $k > 1$.

1.3.3 Fully dynamic maintenance of $k$-connectivity

Another question about the communications network above is to answer the query of whether $u$ and $v$ are in the same $k$-edge/$k$-vertex connected component for a pair of vertices $u$ and $v$ after adding an edge to or deleting an edge from $G$. Assume that this kind of edge insertion or deletion update happens quite often, and we need to respond to such query as soon as possible. Such a problem is usually called the dynamic update
problem or the dynamic maintenance problem.

There have been extensive studies of algorithms for dynamically maintaining the $k$-connectivity property of graphs with $k \leq 4$. For example, Even et al. [50] studied the connected component problem ($k = 1$) in the early of 1980s. Frederickson [54] later studied this problem again, and gave an $O(\sqrt{m})$ fully dynamic sequential algorithm. Eppstein et al. [44] improved Frederickson’s algorithm using their sparsification technique, and presented an $O(\sqrt{m} \log(m/n))$ algorithm. Recently, Eppstein et al. [45] improved their algorithm further by an $O(\log(m/n))$ factor. Moreover, Das et al. [35] and Ferragina [51] developed parallel algorithms with $O(\log n)$ time and $O(n^{2/3} \log \frac{m}{n})$ work for the fully dynamic maintenance of minimum spanning trees and connected components. Independently, Liang et al. [117] also proposed parallel algorithms for these two problems which require $O(\log n \log \frac{m}{n})$ time and $O(n^{2/3})$ processors.

However, as Galil et al. [66] claimed, the fully dynamic maintenance of 2-connectivity is much harder than the maintenance of connected components because the maintenance of connected components only involves merging two connected components into one connected component or splitting a connected component into two connected components per update (an update refers to inserting an edge or deleting an edge), whereas the maintenance of 2-edge/2-vertex connected components may require merging $O(n)$ 2-edge/2-vertex connected components into one or splitting a 2-edge/2-vertex connected component into $O(n)$ 2-edge/2-vertex connected components per update. Westbrook et al. [174] presented the first partially dynamic algorithms for both 2-edge connected components and 2-vertex connected (biconnected) components when inserting edges only. Their algorithms require $O(m\alpha(m,n))$ time for the maintenance of 2-connectivity of graphs when there are $m$ queries and edge insertions. I.e., it costs $O(\alpha(m,n))$ amortized time per update or per query where $\alpha(\ldots)$ is the inverse Ackermann function. Westbrook et al. left an open problem as to whether there exists a sublinear time, fully dynamic sequential algorithm for the 2-connectivity problem. Since then, finding a sublinear time algorithm for this problem has been a challenging task. Combining Frederickson’s clustering technique with other techniques, Galil et al. [66] presented the first sublinear time algorithm for the fully dynamic maintenance of 2-edge connected components. Their algorithm requires $O(m^{2/3})$ time per update. Later Frederickson [55] improved this algorithm to $O(\sqrt{m})$ per update using a so-called ambivalent data structure. Based on their sparsification technique, Eppstein et al. [44, 45] further improved Frederickson’s algorithm by developing $O(\sqrt{m} \log(m/n))$ and $O(\sqrt{m})$ time algorithms respectively. They also derived algorithms for the fully dynamic maintenance of biconnectivity and triconnectivity with $O(\alpha(q,n))$ amortized time per insertion or per query, and $O(n)$ time per deletion, where $q$ is the total number of queries made.

The fully dynamic maintenance of biconnected components seems much harder than that of 2-edge connected components. It can be seen that there is a reduction technique [65] that reduces a $k$-edge connectivity problem into a $k$-vertex connectivity problem, but the reverse reduction has not yet been found. In 1992 Rauch [147] presented the first fully dynamic algorithm for the maintenance of biconnected components. Her algorithm costs $O(m^{2/3})$ amortized time per update and $O(1)$ time per
query. Later she further improved her algorithm to $O(\sqrt{n} \log m)$ by combining new data structures with the sparse certificate technique [148]. Recently Rauch (Henzinger) et al. [81, 80], using a novel decomposition of graphs and randomized technique, suggested polylogarithmic time randomized algorithms for both 2-edge connectivity [81] and biconnectivity [80]. Their algorithms are Las-Vegas type randomized algorithms which are incomparable to the deterministic fully dynamic algorithms, because the deterministic simulation of these randomized algorithms may need an $\Omega(m + n)$ amount of work, due to the fact that the size of the sample space is at least $\Omega(m + n)$ for this case (an excellent book about the topics of randomized algorithms and their deterministic simulations is [134]). Swaminathan et al. [162] also studied the fully dynamic maintenance problem of biconnected components in the distributed computing environment.

When $k = 3$ Galil et al. [67] presented a partially dynamic algorithm for the maintenance of 3-edge connected components when inserting edges only. Their algorithm needs $O((n + q)\alpha(q, n))$ time for any sequence of $q$ queries and updates. Independently, La Poutré et al. [109] also presented a partially dynamic algorithm for this problem with the same time bound as Galil et al. [67] did. Later Galil et al. [64] claimed that they have developed a fully dynamic algorithm for this problem which requires $O(m^{2/3})$ time per update and per query by making minor changes to their approach for 2-edge connectivity [66]. Besides, La Poutré [107] also proposed a partially dynamic algorithm for the maintenance of triconnected components which needs $O((n + q)\alpha(q, n))$ time for any sequence of $q$ queries and updates. As for $k > 3$, there are only a few results available. Kanevskiy et al. [95] presented a partially dynamic algorithm for the maintenance of 4-vertex connected components which requires $O(\alpha(q, n))$ amortized time per update and per query provided the original graph is triconnected where $q$ is the number of operations performed. Dinitz [40] presented a partially dynamic algorithm for the maintenance of 4-edge connected components. Note that combining Eppstein et al.'s [44, 45] sparsification technique and the algorithms above, it is easy to derive the following results: (i) fully dynamic maintenance of 3-edge connected components can be done in $O(q^{2/3})$ time per update and per query; (ii) fully dynamic maintenance of triconnected components can be done in $O(\alpha(q, n))$ time per insertion or per query, and in $O(n)$ time per deletion; (iii) fully dynamic maintenance of 4-vertex connected components can be done in $O(\log n)$ time per insertion, in $O(n \log n)$ time per deletion, and in $O(1)$ time per query.

It is surprising that we have not seen any NC algorithm for the fully dynamic maintenance of the $k$-connectivity problem with fixed $k > 1$, using a sublinear number of processors $o(m + n)$. In particular, we have not seen any NC algorithm for the fully dynamic maintenance of 2-connectivity with at most $O(n)$ processors.

### 1.3.4 The partially dynamic maintenance of the solution of all pairs shortest paths

Suppose that the communications network $G$ is a directed weighted network in which the direction of each edge represents that the communication is single direction, and the weight associated with each edge represents the communication delay on this edge.
When inserting in or deleting an edge from $G$, the following questions are natural: whether $v$ is reachable from $u$ for any pair of vertices $u$ and $v$; what is the length of the shortest path from $u$ to $v$; what is the shortest path between $u$ and $v$, and so on. Usually to cope with these queries on a weighted, directed network is more complicated than that on an unweighted, undirected network.

The all pairs shortest paths problem is a well known problem in graph theory. There are many sequential and parallel algorithms for this problem. But the fully dynamic maintenance to the solution of all pairs shortest paths seems difficult. For a general graph, we have not seen fully dynamic algorithm which is better (in terms of time complexity) than computing the solution from scratch. However, there do exist better algorithms for the fully dynamic update [9, 125] if there are restrictions on edge cost. For example, Ausiello et al. [9] and Lin and Chang [125] considered a special case of this problem in which the edge cost is an integer bounded by $C$. For this case their algorithm requires $O(Cn^3 \log n)$ amortized time for an intermixed sequence of edge insertions and edge's weight decreases, $O(1)$ time for a query of the length between two given vertices, and $O(l)$ time for finding the shortest path between two given vertices $x$ and $y$ where $l$ is the number of edges of the shortest path from $x$ to $y$. For a general graph, Even and Gazit [49] have shown that it requires $O(n^2)$ time for an edge insertion and/or an edge cost decrease, and $O(mn + n^2 \log n)$ time for an edge deletion and/or an edge cost increase, where $m$ is the number of edges in the current graph. Rohmert [151] also considered the fully dynamic maintenance of this problem by giving similar results.

It is interesting to see that the partially dynamic maintenance of the solution of all pairs shortest paths can also be extended to other dynamic update problems such as the all pairs longest paths problem in a directed acyclic graph (DAG), the topological sorting problem on a DAG, and the transitive closure problem in a directed graph.

Though there are so many efficient sequential algorithms for these dynamic update problems, as far as we know, there are no better (in terms of processors used) parallel algorithms for the problems than that to compute the solution from scratch. Note that the well known NC algorithms for the reachability problem need $M(n)$ processors, where $M(n)$ is the number of boolean operations needed to implement $n \times n$ boolean matrix multiplication [70]. Currently $M(n) = O(n^{2.376})$ [32].

1.3.5 The $k$ MVE MST problem and related problems

Consider the communications network $G$ again. We can view it as a distributed computing network in which the high level communication protocol is vital for broadcasting and processors-to-processor communications. Since the minimum connected subgraph of $G$ is an MST of $G$, and an MST of $G$ is often used as the high level communication protocol of the network.

In the following we consider the reliability issue of this high level communication protocol. Assume $G$ is $(k+1)$-edge connected at least. Otherwise, $G$ is disconnected after deleting all edges in a minimum edge cut $S$ and $|S| \leq k$. Without loss of generality, we also assume that the weights of all edges in $G$ are distinct. Then the MST or the
minimum spanning forest (MSF) of $G(V, E - S)$ is unique for any $S \subseteq E$.

Now consider the case of deleting $k$ edges from $G$ such that the MST in the resulting graph has the maximum weight, i.e., the deletion of the $k$ links results in the maximum increase in the communication cost of the high level communication protocol. Thus, the $k$ edges are called the $k$ most vital edges with respect to the MST, and the problem of finding these $k$ edges is called the $k$ MVE MST problem. The study of this problem is very important. We can illustrate its importance from either the positive side or the negative side. From the positive side, consider the current communications network, if an investment budget increases, we try to find $k$ links (edges) in the network which are the most vulnerable, i.e., the failure of the $k$ links will result in the maximum increase in the cost of the high level communication protocol. Therefore we should invest money to the $k$ links to increase their reliability. From the negative side, assume that the communications network is the enemy communications network, if there are $k$ people available currently, and each person can disconnect a link at most, the target is to assign these $k$ people to break the network down. As a result, we expect to have the maximum damage to the network or have the maximum cost to recover the network.

A similar problem is the $k$ most vital edge problem with respect to a shortest path from $s$ to $t$ (the $k$ MVE SP problem for short), which can be defined as follows. Given two vertices $s$ and $t$, delete $k$ edges from $G$ such that the length of the shortest path from $s$ to $t$ in the resulting graph is maximized, assume that there are at least $k + 1$ edge-disjoint paths from $s$ to $t$.

For $k = 1$ this special case, the 1 MVE MST problem becomes that of finding the single most vital edge with respect to an MST of $G$, which has been extensively studied in the literature [83, 88, 93, 157]. Hsu et al. [83] first considered this problem, and gave $O(n^2)$ time and $O(m \log m)$ time sequential algorithms respectively. Iwano et al. [93] further improved Hsu et al.’s results by giving an $O(t_{MST} + \min\{m\alpha(m, n), m + n \log n\})$ time algorithm where $t_{MST}$ is the time used to find an MST of $G$. Hsu et al. [88] later also proposed parallel algorithms for the problem on an EREW PRAM. One of their parallel algorithms requires $O(n^{1+x})$ time and $O(n^{1-x})$ processors with $0 < x < 1$. The other requires $O(m \log (m/N)/N + n\alpha(m, n) \log (m/n))$ time and $O(N)$ processors where $N \leq (m \log m)/\alpha(m, n) \log (m/n))$. Recently Shen [157] presented another parallel algorithm for the problem. His algorithm requires either $O(\log n)$ time and $O(m \log \log n/\log n + n)$ processors on a PRIORITY CRCW PRAM, or $O(\log n \log \log n)$ time and $O(m + n^2/(\log n \log \log n))$ processors on a CREW PRAM.

However, not much work has been done for the $k$ MVE MST problem with arbitrary $k$. The only results are due to Lin et al. [126], Frederickson et al. [58] and Shen [158]. Lin et al. have shown that a generalized version of this problem is NP-complete. In that case each edge in $E$ is assigned a removing cost and the total removing cost $B$ is bounded. Frederickson et al. [58] recently proved that this problem is NP-complete, and presented an approximate algorithm for it. Their approximate algorithm requires $O(\min\{km \log n + k^2 n \log n, km \log^2 n\})$ time, and the solution delivered by their algorithm is $\Omega(1/\log k)$ times optimal. Shen [158] explored this problem by giving a randomized, approximation algorithm. Note that, if $k$ is fixed, there always exists a trivial algorithm running in time $O(m^k T_{MST}(n, m - k))$ for the $k$ MVE MST problem.
where \( T_{MST}(n, m - k) \) is the time used to find the MST of a graph with \( n \) vertices and \( m - k \) edges. However, for fixed \( k > 1 \), no better sequential algorithms have been found, not to mention parallel algorithms.

A number of researchers have made some contributions to the 1 MVE SP problem in the past decade. Corley et al. [33] first raised this problem by giving some preliminary results. Malik et al. [130] later presented an efficient algorithm when \( G \) is an undirected graph. Unfortunately, their algorithm is incorrect, as shown by Bar-Noy et al. [16] who gave a counter-example. The generalization of this problem is the \( k \) MVE SP problem [15, 16, 33, 124]. Corley et al. stated in [33] that the identification of the sufficient conditions leading to a constructive procedure to solve the \( k \) MVE SP problem is an open problem. Ball et al. [15] generalized this problem by assigning a removal cost to each edge, and the total removal cost is bounded. For this general case, they proved that the problem is NP-complete. However, their proof does not imply that the \( k \) MVE SP problem is NP-complete. Bar-Noy et al. [16] have shown that the \( k \) MVE SP problem is NP-complete too. Recently, Liang et al. [124] gave a fast polynomial algorithm for the \( k \) MVE SP problem when \( k \) is fixed.

1.3.6 The approximate \( t \)-spanner problem

Considering the communications network \( G \) again, we already mentioned that an MST of \( G \) is usually used as the high level communication protocol of the network because it is the minimum connected subgraph of \( G \). Therefore, using it as a communication protocol is communication cost optimal. However, this communication protocol is vulnerable as can be seen from the above discussion. Besides that, the communication time is not as good as it should be (assume the communication time between two neighboring processors is unit time). Consider the following case: a ring consists of \( n \) processors. The MST of this ring is obviously a line, and the communication time between the processors at the beginning and at the end of this line takes \( n - 1 \) unit times. So, from the transfer speed point of view, this is not what we expected. Thus we need to design a protocol which has a trade-off between the communication cost and the time used for communications. The spanner [3, 141] is a good choice to implement this kind of trade-off. Furthermore, the spanner concept has been found to have other applications. For example, Peleg and Upfal [143], Awerbuch et al. [11], and Awerbuch and Peleg [12] use it to design efficient routing schemes in distributed networks. Peleg and Ullman [142] use it to design synchronizers [10]. Cohen [29] uses it to find the approximate all pairs shortest paths by giving a randomized parallel algorithm for finding a sparse \( t \)-spanner with size \( O(n^{1 + \frac{1}{12}}) \) on a weighted graph which needs \( O\left(\frac{W_{\text{max}}}{W_{\text{min}}} t^{2 \log^2 n} \right) \) expected time and \( O(n^{1/2} m \beta \log^2 n \) work on an EREW PRAM, where \( \beta = \frac{t_t}{2 + \epsilon / 2} \), where \( wt(e) \) is the weight of \( e \), \( W_{\text{max}} = \max\{wt(e) \mid e \in E\} \), \( W_{\text{min}} = \min\{wt(e) \mid e \in E\} \), and \( \epsilon \) is an arbitrarily small constant.

Much effort for finding sparse \( t \)-spanners on some special graphs such as Euclidean graphs, geometry graphs and chordal graphs has been taken in several recent works [5, 20, 36, 37, 141, 167]. Peleg and Schäffer [141], Althöfer et al. [3] have shown some very interesting properties of \( t \)-spanners of unweighted graphs and weighted graphs. Despite
the existence of several efficient sequential and distributed algorithms for finding a sparse \(t\)-spanner of graphs, we have not seen any deterministic parallel algorithm for this problem. In this thesis we first relax the restriction of the problem by introducing the approximate \(t\)-spanner concept, and then present simple parallel algorithms for finding a sparse, approximate \(t\)-spanner on both unweighted and weighted graphs in terms of size and weight. The algorithms exhibit some trade-offs between the running time and the factor of the spanner generated.

### 1.3.7 The edge-coloring problem and related problems

It is well known that vertex- and/or edge-coloring is very useful to solving the scheduling problems such as class table and job scheduling, etc. In the parallel and distributed computing environments, it is widely used to solve the source conflict problem incurred by parallelism.

Let \(G(V, E)\) be a simple graph, an edge coloring of \(G\) is an assignment of colors to its edges such that all edges sharing a common vertex are assigned different colors. Furthermore, we wish to use as few colors as possible to color \(G\). The minimum number of colors used is called chromatic index, denote by \(\chi'(G)\). Vizing showed that \(\Delta \leq \chi'(G) \leq \Delta + 1\) [171]. In fact, Vizing’s proof implies \(O(mn)\) sequential time algorithm with \(\Delta + 1\) colors to color all edges of \(G\). However, Holyer has shown that deciding whether a graph requires \(\Delta\) or \(\Delta + 1\) colors is NP-complete, even when restricted to the class of cubic graphs [82]. Therefore the edge-coloring problem here is to edge-color \(G\) with \(\Delta + 1\) colors. One variety of this problem is the edge-coloring update problem which can be described as follows. Let \(G\) be an already edge-colored graph with \(\Delta + 1\) colors, consider add a new vertex \(v\) and the edges incident to \(v\) into \(G\), re-color the edges in this resulting graph with these \(\Delta + 1\) colors provided the number edges incident to \(v\) is bounded by \(\Delta\). The other related problem is the approximate edge-color problem, for which we use more than \(\Delta + 1\) colors instead of \(\Delta + 1\) colors exactly to edge-color \(G\). This approximate edge-coloring abides by the practical philosophy that in many cases the exact solution is not necessary, but an approximate solution is acceptable.

A number of parallel algorithms exist for the edge-coloring problem on some special graphs such as bipartite graphs [111], trees, outplanar graphs, Halin graphs [71, 72], and planar graph [28, 27, 79]. For multigraphs, Upfal presented an \(O(\log^3 n\Delta)\) time algorithm with \(3[\Delta/2]\) colors by using \(O(n\Delta)\) processors (appears in [96]). For the special class of multigraphs of \(\Delta = 3\), Karloff and Shmoys [96] presented an algorithm which runs in \(O(\log n)\) time and uses \(O(n)\) processors. For general simple graphs, the pioneering work is due to Karloff and Shmoys [96]. They presented an edge-coloring parallel algorithm with \(\Delta + 1\) colors that requires \(O(\Delta^6 \log^4 n)\) time and \(O(n^2 \Delta)\) processors on a COMMON CRCW PRAM, assume that the fastest known algorithm for finding maximal independent sets [74] is employed. They also presented a randomized \(NC\) parallel algorithm with \(\Delta + 20\Delta^{1/2 + \epsilon}\) colors for edge-colorings, where \(\epsilon \leq 1/4\), which runs in \(O(\log O(1) n)\) expected time and uses \(O(n O(1))\) processors (independent of \(\epsilon\)).
1.3.8 The maximal interlocking set problem

Given a family of $n$ intervals $I$ on the real line, we say that two intervals $I_i = [a_i, b_i]$ and $I_j = [a_j, b_j]$ are interlocked if and only if either $a_i < a_j < b_i < b_j$ or $a_j < a_i < b_j < b_i$. The problem is to find all maximal interlocking sets in $I$. This problem has practical application in VLSI layout. For example, the determination of interlocking sets is used in single-row routings [145, 153]. There, a set of two-point nets lying in a single row is given, the goal is to find a routing of these nets that minimizes the maximum channel width provided the routing exists. The solutions to this routing problem are given in [145, 153]. A key component of these solutions is the partitioning of the nets into maximal interlocking sets.

There are some studies regarding this problem, Raghavan and Shani [145] presented an $O(n^2)$ time algorithm using dynamic programming technique. After investigating the property of this special graph, Lloyd [127] presented an $O(n \alpha(2n, n))$ time sequential algorithm provided the endpoints of intervals are sorted initially. However, his algorithm seems highly sequential. Later, Saxena and Prasad [153] studied this problem again by presenting a parallel algorithm that requires either $O(\log^2 n)$ time and $O(n)$ processors on a CREW PRAM, or $O(\log n \log \log n)$ time and $O(n)$ processors on an ARBITRARY CRCW PRAM. Meanwhile, they also gave an $O(n)$ time sequential algorithm provided the endpoints of intervals are integers, with the fact that $n$-integer sorting can be done in $O(n)$ time. Otherwise, their sequential algorithm still needs $O(n \log n)$ time.

1.4 Overview of the Thesis

In this thesis we consider a number of graph problems in the parallel setting. These include the fully dynamic maintenance to $k$-edge/$k$-vertex connectivity with $k = 2, 3$; the partially dynamic maintenance to the solution of all pairs shortest paths when edge insertion operations are allowed only; testing $k$-connectivity in a directed graph with fixed $k$; augmenting a graph into $k$-edge or $k$-vertex connected by a given feasible set of weighted edges at optimal cost with $k = 2$; finding the $k$ most vital edges with respect to an MST with fixed $k$; finding approximate sparse, $t$-spanners on weighted/unweighted graphs; edge-coloring a simple graph; and finding all maximal interlocking sets in a family of intervals. Even though these problems seem diverse, they share at least one of the following properties: (i) the problems involve dynamic updates (chapters 2, 3, 6, 7 and 9); (ii) the sparsification technique is applicable to the problems (chapters 2, 4, 6, 7 and 10); or (iii) the problems are related to reliability issues of communications networks, e.g. network connectivity testing, connectivity augmentation, and routing reliability etc (chapters 2, 3, 4, 5, 6, 7 and 8). In summary, this thesis has achieved the following results.

Chapter 2 presents an NC algorithm for the fully dynamic maintenance of $2$-edge connectivity of undirected graphs. Our algorithm requires $O(\log^2 n)$ time and $O(n^{3/4})$ processors per update and per query on a PRIORITY CRCW PRAM which is the
first NC algorithm with $o(n)$ \footnote{$g(n) = o(f(n))$ if $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$, where $f(n) > 0$ and $g(n) > 0$.} work for this problem. Incorporating with the sparsification technique of Eppstein et al. [44, 45], we extend our results to the other graph connectivity problems such as biconnectivity, 3-edge connectivity, and triconnectivity by presenting NC algorithms for each of them. The fully dynamic maintenance version of these problems can be done with $\tilde{O}(n)$ \footnote{$\tilde{O}(f(n)) = O(f(n) \log^c n)$, where $c$ is constant.} work, in contrast with $\Omega(m + n)$ work from scratch \cite{164, 61}. Two key techniques are involved. One is to partition the graph into vertex clusters, and to find a way to maintain a succinct encoding of each cluster such that each cluster satisfies some nice properties, for example, keeping its size under a specific size control, and answering the query on it or between different clusters as quick as possible. The other is to employ the graph decomposition concept \cite{45} in which the original graph $G$ is decomposed into $[m/n]$ subgraphs initially, then apply the first key technique to each of the subgraphs. The aim is to answer the queries on $G$ by only retrieving some of the subgraphs. In doing so, some new data structures for parallel processing are needed. This enables us to answer the queries about the edge/vertex connectivity properties without checking the entire graph each time.

Chapter 3 presents an NC algorithm for the partially dynamic maintenance of all pairs shortest paths on a directed weighted graph after undergoing a sequence of edge insertions. Our algorithm for each of such updates requires $O(\log n)$ time and $O(n^2)$ processors on a CREW PRAM. By a minor modification of this algorithm, we can also solve the longest path problem on a DAG, the topological sorting problem in a DAG, and the transitive closure (the reachability) problem on a directed graph with the same time and processor bounds under the restriction of edge insertions only. This is obtained by maintaining the shortest path tree starting at each vertex and by developing sophisticated algorithms for updating the trees efficiently. Our basic approach to the partially dynamic maintenance of all pairs shortest paths is to reduce it to a number of tree update problems, then we further reduce the tree update problem to a connected component problem in an undirected graph. It is interesting to know how to implement the reduction because these two problems seem no relationship at all in sequential, the former is a problem on a directed weighted graph, and the latter is a problem on an undirected graph.

Chapter 4 presents NC algorithms for answering the query of whether $G$ is $k$-edge/$k$-vertex connected for a given directed and/or undirected graph $G$ where $k = O(\log^c n)$ and $c$ is a constant. The technique employed is the well known elementary method. As a result, we establish the NC membership of the problem on directed graphs.

Chapter 5 deals with the optimal $k$-connectivity augmentation problem on $G(V, E_0)$ by presenting NC approximation algorithms for it. Our contributions can be described as follows. We devise (i) an NC approximation algorithm for the optimal 2-edge connectivity augmentation problem which delivers a solution either within $(1 + \ln n_e)(1 + \epsilon)$ times optimum if $G$ is connected, or within $(1 + \ln n_e)(1 + \epsilon) + 1$ times optimum otherwise; (ii) and an NC approximation algorithm for the optimal biconnectivity augmentation problem which delivers a solution within either $(1 + \ln n_b)(1 + \epsilon) \log n_b$ times optimum if
$G$ is connected, or within $(1 + \ln n_b)(1 + \epsilon) \log n_b + 1$ times optimum otherwise, where $n_c$ and $n_b$ are the number of 2-edge connected components and biconnected components of $G(V, E_b)$ respectively, and $\epsilon$ is a small constant with $0 < \epsilon < 1$. We also present an approximation NC algorithm for the optimal $k$-edge connectivity augmentation problem with fixed $k > 2$. For this latter case, if $G = (V, E_b)$ is $(k - 1)$-edge connected, our NC algorithm delivers an approximation solution which is either $\beta t$ times optimum if $k$ is odd or $\beta$ times optimum otherwise, where $\beta = (1 + \ln n)(1 + \epsilon), 1 \leq t \leq \lceil \log_2 n \rceil$, and $\epsilon$ is a constant with $0 < \epsilon < 1$. If $G$ is $\lambda$-edge connected and $1 \leq \lambda < k - 1$, our NC approximation algorithm delivers a solution within $\alpha \beta$ times optimum where $\beta = (1 + \ln n)(1 + \epsilon)$, $\epsilon$ is a constant with $0 < \epsilon < 1$, and $\alpha$ is defined as follows: (i) if both $k$ and $\lambda$ are odd (even), then $\alpha = \frac{(k-\lambda)}{2} \lceil t + 1 \rceil$; (ii) if $k$ is even and $\lambda$ is odd, then $\alpha = t \lceil \frac{k-\lambda}{2} \rceil + \lceil \frac{k-\lambda}{2} \rceil$; (iii) otherwise, $\alpha = t \lceil \frac{k-\lambda}{2} \rceil + \lceil \frac{k-\lambda}{2} \rceil$, where $1 \leq t \leq \lceil \log_2 n \rceil$. As a consequence of the results above, there exists an NC approximation algorithm for finding a minimum weighted $k$-edge connected subgraph on a $k$-edge connected weighted graph, delivering a solution at most $\alpha \beta$ times optimum, where $\beta = (1 + \ln n)(1 + \epsilon)$, $\epsilon$ is a constant with $0 < \epsilon < 1$, and $\alpha$ is defined as follows: $\alpha = \frac{k t + k - \lambda t + 1}{2}$ if $k$ is odd, $\alpha = t \lceil \frac{k-\lambda}{2} \rceil + \lceil \frac{k-\lambda}{2} \rceil + 1$ otherwise, where $1 \leq t \leq \lceil \log_2 n \rceil$. The proposed NC algorithms for the above problems are the first NC algorithms giving these degrees of approximation. The basic technique adopted is to reduce the problems into the minimum weighted set cover problem for which there are NC approximation algorithms available [17, 146].

Chapter 6 presents efficient sequential and parallel algorithms for the $k$ MVE MST problem with fixed $k$. In particular for $k = 1$ this special case, we develop the most efficient NC algorithms for this problem on both the EREW PRAM and the CREW PRAM so far. The algorithm on the EREW PRAM runs in $O(\log n \log \log n)$ time with $O(m + n^2 / \log n)$ processors which is the fastest NC algorithm for this problem on this model; the algorithm on the CREW PRAM runs in $O(\log n \log \log n)$ time with $O(m + n)$ processors which is the best in the sense of work so far. For the case of $k = 2, 3$, we devise sequential algorithms for them. The time complexities of the proposed algorithms are $O(n^2 \alpha(3n, n))$ time and $O(n^3 \alpha(4n, n))$ respectively. By parallelizing these two sequential algorithms, we derive two NC algorithms which require $O(\log n \log \log n)$ time using $O(n^2 / \log n)$ processors and $O(\log n \log \log n)$ time using $O(n^3 / \log n)$ processors on a CREW PRAM. For a fixed $k \geq 4$, we first present an $O(n^{k+1})$ time sequential algorithm, and then develop an NC algorithm from this sequential algorithm. The NC algorithm requires $O(\log n + T_{MST}(n, m))$ time and $O(n^{k+1})$ processors on an EREW PRAM, where $T_{MST}(n, m)$ is the time used to find an MST in a graph with $n$ vertices and $m$ edges on an EREW PRAM using polynomial processors. Currently the best result for $T_{MST}(n, m) = O(\log n \log \log n)$ and the number of processors is $O(m + n)$ [26]. We achieve the results above by extending the sparse $k$-edge certificate concept on an unweighted undirected graph [135, 136] to the sparse, weighted $k$-edge certificate on a weighted undirected graph. Note that the construction of sparse, weighted $k$-edge certificates is different from the construction of sparse $k$-edge certificates. We later show that the $k$ MVE MST problem on $G$ is equivalent to the $k$ MVE MST problem on its sparse, weighted $(k + 1)$-edge certificate.
We thus obtain efficient sequential and parallel algorithms for this problem by using the sparse weighted \((k + 1)\)-edge certificate. We believe, the sparse, weighted \(k\)-edge certificate by itself may be found useful to solve other graph problems. Note that the sparse, weighted \(k\)-edge certificate of \(G\) is a sparse subgraph of \(G\). This further demonstrates that the sparsification technique is very powerful in the design of efficient sequential and parallel graph algorithms.

Chapter 7 presents parallel implementation algorithms for the 1 MVE MST problem and the 1 MVE SP problem on undirected graphs on the popular parallel computers - meshes and hypercubes. This chapter involves the implementation issue of algorithms of PRAMs. That is, how to simulate a PRAM algorithm in a practical interconnection network of processors. We first simulate our 1 MVE MST algorithm of the PRAM in Chapter 6 on the meshes by utilizing the special topological structure of meshes. The major concern of this simulation is to design an efficient routing subroutine which is used to implement each step of the PRAM algorithm. As a result, we obtain a parallel algorithm for this problem which requires \(O(n)\) time and \(n^2\) processors. Note that the algorithm on this model is the first one which is also time-optimal because the trivial communication lower bound of this model is \(Ω(n)\). Then we show how to simulate each step of the mesh algorithm on the hypercube. We thus obtain an NC algorithm for this problem in \(O(\log^2 n)\) time using \(n^3\) processors on a hypercube. We solve the 1 MVE SP problem on an undirected weighted graph by reducing it to an 1 MVE MST problem first, then following the discussion above, we can solve this latter problem with the same time and processor bounds as that for the 1 MVE MST problem on the meshes and hypercubes respectively. It must be said that in the sequential environment this is not a good idea to reduce an 1 MVE SP problem to an 1 MVE MST problem because it may take more time to implement this reduction than that to find the solution directly. However, because of the special structure of meshes, this enables us to implement this reduction on a 2-D mesh model with the optimal time bound.

Chapter 8 presents parallel algorithms for finding approximate \(t\)-spanners on undirected weighted/unweighted graphs. The algorithm exhibits certain trade-offs between the running time and degree of approximation. In detail, if \(G\) is an unweighted graph, our algorithm requires \(O(\frac{n}{T}\log n)\) time and \(M(n)\) processors, and the spanner generated has size of \(O\left(\left(\frac{n}{T}\right)^{1+\frac{1}{t}} + n\right)\) and factor of \(O(D^{k+1})\); otherwise our algorithm requires \(O\left(\left(\frac{n}{T}\right)^{1+\frac{1}{t}} + \left(\frac{n}{T}\right)^{1+2/(t-2)} \log n\right)\) time and \(O(n^2)\) processors, and the spanner generated has size of \(O\left(\left(\frac{n}{T}\right)^{1+\frac{1}{t-1}} + n\right)\), weight of \(\left(\frac{n}{T}\right)^{\frac{2}{t-1} + 1} wt(MST)\), and factor of \(O(D^{k+1})\), where \(M(n)\) is the number of processors needed to find a Breath-First Search tree in a graph with \(n\) vertices in time \(O(\log n)\), \(D\) is the maximum edge weight of an MST of \(G\), and \(k\) is a fixed integer with \(1 \leq k \leq \log n\). Also, an NC algorithm for finding a \(2t\)-spanner on a weighted graph \(G\) with size \(O(\min\{m, \frac{n}{T}\log_2+e(\frac{W_{\max}}{W_{\min}})\})\) is presented. The proposed algorithm requires \(O(\log^3 n \log_2 + e(\frac{W_{\max}}{W_{\min}}))\) time and \(O(n^3)\) processors, where \(W_{\max} = \max\{wt(e) \mid e \in E\}\), \(W_{\min} = \min\{wt(e) \mid e \in E\}\), and \(e\) is a constant with \(0 < e < 1/2\). All algorithms run in a CRCW PRAM model.

Chapter 9 presents an improved parallel algorithm for the edge-coloring problem. This problem is totally different from the problems in the previous chapters. However,
the reason for including it in this thesis is that its solution depends on the result of the edge-coloring update problem which belongs to the framework of dynamic update. For the edge-coloring update problem we present a parallel algorithm to implement this update in $O(\Delta^{3/2} \log^2 \Delta + \Delta \log n)$ time using $O(n \Delta + \Delta^3)$ processors on a COMMON CRCW PRAM. Then using this algorithm as a subroutine, incorporating with another technique, we presents the fastest parallel algorithm for the edge-coloring problem so far. Our algorithm for the edge-coloring problem needs $O(\Delta^4 \log^4 n + \Delta^{4.5} \log^3 \Delta \log n)$ time and $O(n^2 \Delta + n \Delta^3)$ processors, improving the best known algorithm by an $O(\Delta^{1.5})$ factor in time. Finally we extend the edge-coloring problem to the approximate edge-coloring problem, for which we present a parallel algorithm whose running time depends on the color accuracy, i.e., the more colors used, the less time spent. It exhibits some trade-offs between the time spent and the number of colors available. The technique involved for the edge-coloring update problem is how to make the “fan”-coloring operation that seems highly sequential become parallelizable. We implement this by building an auxiliary graph, and running a parallel depth-first search algorithm on this auxiliary graph. Apart from that, we adopt the following strategy in the design of our algorithms, i.e., we use more than $\Delta + 1$ (actually $3\lceil \Delta/2 \rceil$) colors to edge-color $G$ initially, then remove all extra colors phase by phase until there are $\Delta + 1$ colors left only and all edges are colored by the remaining colors, whereas all previously known parallel algorithms to edge-color $G$ also proceed phase by phase but they never use more than $\Delta + 1$ colors in any phase. Using this technique and combining the results of the edge-coloring update algorithm, we obtain the fastest parallel algorithm for this problem so far.

Chapter 10 presents a fast NC algorithm for finding all maximal interlocking sets in a family of intervals. Our algorithm requires $O(\log n \log \log n)$ time and $O(n)$ processors on an EREW PRAM which is the most efficient until now. This problem can be formed as a connected component problem on an interval graph $G$ based on the interval intersection relationship. Here we use the sparsification technique in a non-trivial way. Usually a sparse subgraph is obtained from its original graph. In this special case, we yield the sparse subgraph without construction of the original graph. The technique involved is to construct a very sparse subgraph $G'$ ($2n - 2$ edges at most) and to use “the all nearest smaller (larger) values problem” as a primitive. As a result, we compute all connected components of $G'$ rather than $G$ to find all connected components of $G$. We thus find all maximal interlocking sets of a family of intervals.

In this chapter we have attempted to give a brief summary of the results obtained in this research, and to place our results in context with the previously known related results. In order to enhance readability, we make each chapter self-contained. Thus, in certain cases there will be some repetition between chapters, in particular the recapitulation of current literature is repeated in some chapters.
NC Algorithms for the Fully Dynamic Maintenance of $k$-Connectivity with $k = 2, 3$

2.1 Introduction

Let $G(V, E)$ be a simple undirected graph, the $k$-connectivity of $G$ is a basic property in graph theory with wide applications in augmented compilers, distributed computing, reliable communications networks, etc. There have been extensive studies of algorithms for dynamically maintaining the $k$-connectivity property of graphs. For example, Even et al. [50] studied the connected component (CC) problem in the early of 1980s. Later Frederickson [54] studied a general version of this problem — the MST problem, by presenting an $O(\sqrt{n})$ fully dynamic algorithm. Frederickson’s algorithm is based on the concept called multi-level topological partition of the set of vertices, i.e., the clustering partition concept. Eppstein et al. [44] improved Frederickson’s algorithm using a technique called sparsification, and presented an $O(\sqrt{n} \log(m/n))$ algorithm. Later Eppstein et al. [45] improved their algorithm further by an $O(\log(m/n))$ factor. However, all the algorithms above are hardly parallelizable since they use Frederickson’s restricted clustering decomposition which seems highly sequential. Recently, Das et al. [35] and Ferragina [31] developed NC algorithms with $O(\log n)$ time and a total of $O(n^{2/3} \log \frac{m}{n})$ work for the fully dynamic maintenance of MSTs and CCs. Independently, Liang et al. [117] also proposed an NC algorithm for them which require $O(\log n \log \frac{m}{n})$ time and $O(n^{2/3})$ processors.

However, as Galil et al. [66] pointed out, the fully dynamic maintenance of 2-edge/2-vertex connected components (2ECCs/2VCCs) is much harder than the maintenance of CCs because the maintenance of CCs only involves merging two CCs into one CC or splitting a CC into two CCs per update, whereas the maintenance of 2ECCs/2VCCs may require merging $O(n)$ 2ECCs/2VCCs into one 2ECC/2VCC or splitting a 2ECC/2VCC into $O(n)$ 2ECCs/2VCCs per update. Westbrook et al. [174] presented the first partially dynamic algorithms for both 2ECCs and 2VCCs. Their algorithms require $O(m \alpha(m, n))$ time for the maintenance of these properties of graphs when there are $m$ queries and edge insertions. i.e., it costs $O(\alpha(m, n))$ amortized time.
per update or per query. Westbrook et al. left an open problem as to whether there exists a sublinear time, fully dynamic algorithm for the 2-connectivity problem. Since then, finding a sublinear time algorithm for this problem has been a challenging task. Combining Frederickson's clustering technique with other techniques, Galil et al. [66] presented the first fully dynamic algorithm with sublinear time for the maintenance of 2ECCs which requires $O(m^{2/3})$ time per update. Later Frederickson [55] improved this algorithm to $O(\sqrt{m})$ using a so-called ambivalent data structure. Based on their sparsification technique, Eppstein et al. [44, 45] further improved Frederickson's algorithm by developing $O(\sqrt{n \log(m/n)})$ and $O(\sqrt{n})$ time algorithms. They also derived algorithms for the fully dynamic maintenance of biconnectivity and triconnectivity of a graph which need $O(\alpha(q,n))$ amortized time per insertion or per query, and $O(n)$ time per deletion, where $q$ is the total number of queries made.

Generally speaking, the fully dynamic maintenance of 2VCCs (or biconnected components) seems much harder than that for 2ECCs. It can be seen that there is a reduction technique [65] which can reduce a $k$-edge connectivity problem into a $k$-vertex connectivity problem, but the reverse reduction has not yet been found. In 1992 Rauch [147] presented the first fully dynamic algorithm for the maintenance of 2VCCs which costs $O(m^{2/3})$ amortized time per update and $O(1)$ time per query. Later she further improved her algorithm to $O(\sqrt{m \log m})$ by combining new data structures with the sparse certificate technique [148]. Recently (Rauch) Henzinger et al. [81, 80], using a novel decomposition of graphs and randomized technique, suggested polylogarithmic time randomized algorithms for both 2ECCs [81] and 2VCCs [80]. But their algorithms are Las-Vegas type randomized algorithms which are incomparable to the deterministic algorithms, because the deterministic simulation of these randomized algorithms may need an $\Omega(m + n)$ amount of work. It is worthy to mention that Swaminathan et al. [162] also studied the fully dynamic maintenance of 2VCCs in the distributed computing environment.

When $k = 3$, Galil et al. [67] presented a partially dynamic algorithm for the maintenance of 3-edge connected components (3ECCs) when inserting edges only. Their algorithm needs $O((n + q)\alpha(q,n))$ time for any sequence of $q$ queries and updates. Independently, La Poutré et al. [109] also presented a partially dynamic algorithm for this problem with the same time bound as Galil et al. did. Later Galil et al. [64] claimed that by making minor changes of their method for 2ECCs [66], they developed a fully dynamic algorithm for 3ECCs which requires $O(m^{2/3})$ time per update and per query. Besides above, La Poutré [107] also proposed a partially dynamic algorithm for the maintenance of triconnected components (3VCCs) which needs $O((n + q)\alpha(q,n))$ time for any sequence of $q$ queries and updates. As for $k > 3$, there are only a few results for maintaining $k$-connectivity. Kanevsky et al. [95] presented a partially dynamic algorithm for the maintenance of 4-vertex connected components which requires $O(\alpha(q,n))$ amortized time per update and per query provided the original graph is triconnected, where $q$ is the number of operations performed. Dinitz [40] presented a partially dynamic algorithm for the maintenance of 4-edge connected components. Note that combining Eppstein et al.'s [44, 45] sparsification technique and the algorithms above, it is easy to derive the following results: (i) fully dynamic maintenance
of 3ECCs can be done in $O(n^{2/3})$ time per update and per query; (ii) fully dynamic maintenance of 3VCCs can be done in $O(\alpha(q, n))$ time per insertion or per query, and in $O(n)$ time per deletion; (iii) fully dynamic maintenance of 4-vertex connected components can be done in $O(\log n)$ time per insertion, in $O(n \log n)$ time per deletion, and in $O(1)$ time per query.

In this chapter we consider how to maintain $k$-edge and $k$-vertex connectivity of a graph with $k = 2, 3$ in parallel after undergoing an intermingled sequence of insertion, deletion and query operations described as follows.

- \textit{SameConnComp}(x, y): return true if vertices $x$ and $y$ are in the same $k$-edge or $k$-vertex connected component. Otherwise return false.

- \textit{InsertEdge}(x, y): insert a new edge between $x$ and $y$.

- \textit{DeleteEdge}(x, y): delete the edge between $x$ and $y$.

We present NC algorithms for the fully dynamic maintenance of $k$-edge/$k$-vertex connected components of $G$ with $k = 2, 3$ by using at most $O(n)$ processors in contrast to the usual $\Omega(m + n)$ processors needed for computing these properties from scratch. Our major results are listed as follows. (i) As for the 2-edge connectivity problem, we present an NC algorithm which can implement both per update and per query in $O(n \log n \log(m/n))$ time with $O(n^{3/4})$ processors. The basic idea of our algorithm comes from Galil et al.'s sequential algorithm, but it should be mentioned that their algorithm by itself is not easily parallelizable. Our NC algorithm is achieved by using highly parallelizable approaches to replace some parts in their algorithm and collaborating some parallel algorithm techniques. (ii) For the biconnectivity problem, our fully dynamic algorithm requires either $O(n^2)$ time using $O(n(2n, n) / \log n)$ processors per update, and $O(1)$ time with a single processor per query, or $O(n \log n \log(m/n))$ time using $O(n(2n, n) / \log n)$ processors per update and $O(n \log n)$ time with $O(n(2n, n) / \log n)$ processors per query. (iii) By generalizing the method used for biconnectivity, we also derive an NC algorithm for the fully dynamic maintenance of triconnectivity. The algorithm for this latter case requires $O(n \log n \log(m/n) + \log n \log \log n / \alpha(3n, n))$ time using $O(n(3n, n) / \log n)$ processors per update, and $O(1)$ time with a single processor per query. By using the reduction technique of Galil et al. [65] which reduces a $k$-edge connectivity problem into a $k$-vertex connectivity problem, we obtain an NC algorithm for the fully dynamic maintenance of 3ECCs. The algorithm has the same time and processor bounds as that for the fully dynamic maintenance of triconnectivity.

To the best of our knowledge, the proposed NC algorithms are the first NC algorithms for these problems using $O(n)$ processors in contrast to $\Omega(m + n)$ processors for solving them from scratch. In particular, the proposed NC algorithm for the 2-edge connectivity problem uses $\alpha(n)$ processors only. The parallel computational model used in this chapter is a PRIORITY CRCW PRAM.

The remainder of this chapter is organized as follows: in Section 2.2 we introduce some basic concepts with respect to $k$-edge/$k$-vertex connectivity update. Eppstein et al.'s sparsification technique [44] is also introduced in this section. In Section 2.3 we deal with the fully dynamic maintenance of 2ECCs on a connected graph. In Section 2.4 we
first extend our results on connected graphs to disconnected graphs, then collaborating Eppstein et al's sparsification technique, we present an improved algorithm for 2-edge connectivity on general graphs. We also discuss the fully dynamic maintenance of 3-edge connectivity in this section. The fully dynamic maintenance of 2VCCs and 3VCCs is discussed in Section 2.5. We conclude our discussion in Section 2.6.

2.2 Preliminaries

2.2.1 Basic concepts

Here we introduce a technique of transforming a general graph with arbitrary degree into a graph with bounded degree. It is well known [78] that each graph $G(V, E)$ can be transformed into a graph $G'(V', E')$ whose vertices have degree no greater than three. This transformation is described as follows: for each vertex $u$ of degree $d \geq 4$ in $G$, where $v_0, v_1, \ldots, v_{d-1}$ are the adjacent vertices of $u$, replace $u$ with the new vertices $u_0, u_1, \ldots, u_{d-1}$, add edges $(u_i, u_{i+1} \text{ mod } d)$, and replace the edge $(u_i, v_i)$ with edge $(u_i, v_j)$. It is not hard to prove that $G'$ contains at most $O(m + n)$ edges and vertices, and each vertex has degree no greater than three. In this chapter when dealing with the 2-edge connectivity, we assume that $G(V, E)$ is a graph with $O(m + n)$ vertices in which each vertex has degree no greater than three. It is easy to prove that this transformation preserves the 2-edge connectivity of the original graph [66], and the transformation can be implemented in $O(\log n)$ time using $O(m + n)$ processors.

2.2.2 The sparse $k$-edge ($k$-vertex) certificate

Let $G(V, E)$ be an arbitrary graph. A sparse $k$-edge certificate of $G$ is a subgraph $H$ of $G$ that has $O(kn)$ edges and such that every edge separator of size less than $k$ in $H$ is also an edge separator in $G$. A sparse $k$-vertex certificate of $G$ is defined similarly. In the following we present approaches to construct the sparse $k$-edge/$k$-vertex certificate of $G$.

Let $T'_1$ be a maximal spanning forest of $G$, and $T'_i$ be a maximal spanning forest of graph $G_i = G - \cup_{j=1}^{i-1} T'_j$ for $i > 1$. Denote by $U'_i = \cup_{j=1}^{i-1} T'_j$, the union of the maximal spanning forests $T'_1, T'_2, \ldots, T'_i$. Then the graph $U'_k$ is a sparse $k$-edge certificate of $G$ [135, 136], and $U'_k$ has the following property.

**Lemma 2.1** [135, 136] The graph $U'_k$ defined above is $l$-edge connected if and only if $G$ is $l$-edge connected at least, for any integer $l$ with $0 \leq l \leq k$.

Notice that Lemma 2.1 always holds no matter whether $G$ is a simple graph or not. Next we introduce the scan-first search technique [21] on undirected graphs which is defined as follows.

**Definition 2.1** [21] A scan-first search in a connected undirected graph $G$ starting from a specified vertex $v$ is a systematic way of visiting the vertices in $G$. To scan a vertex is to visit all previously unvisited neighbors of that vertex. At the beginning of
the search, only \( r \) is visited. Then, the search iteratively scans an already visited but unscanned vertex until all vertices are scanned.

If \( G \) is disconnected, we obtain a scan-first search forest of \( G \) by applying scan-first search on each CC. Cheriyan et al. [21] presented an efficient parallel algorithm for finding a scan-first search forest. Their algorithm is described as follows. Assume \( G \) is connected. First find an arbitrary spanning tree \( T \) of \( G \); then label each vertex \( T \) by its pre-order label; finally the scan-first tree \( ST \) is defined as: let \( b(v) \) be the parent of \( v \) in \( ST \) and \( N(v) \) be the neighbor sets of \( v \) in \( G \). Then for each vertex \( v, v \neq r, b(v) \) is the vertex in \( N(v) \) that has the smallest label.

**Lemma 2.2** [21] For an undirected graph with \( n \) vertices and \( m \) edges, a scan-first search spanning forest can be found in \( O(\log n) \) time using \( C(n,m) \) processors on a CRCW PRAM, where \( C(n,m) \) is the number of processors used to compute a spanning tree in each CC in \( O(\log n) \) time. Currently the best result of \( C(n,m) \) is \( O((m + n)\alpha(m,n)/\log n) \) [31].

**Theorem 2.1** [21] Let \( G = (V,E) \) be an undirected graph with \( |V| = n \) and \( k \) be a positive integer. For \( i = 1, 2, \ldots, k \), and let \( E_i \) be the edge set of a scan-first search forest in the graph \( G_{i-1} = (V,E - (E_1 \cup \ldots \cup E_{i-1})) \). Then \( C_k = E_1 \cup \ldots \cup E_k \) is a certificate for \( k \)-vertex connectivity of \( G \), and this certificate has at most \( k(n-1) \) edges.

**Lemma 2.3** [21] The graph \( C_k \) defined above is \( l \)-vertex connected if and only if \( G \) is \( l \)-vertex connected at least, for any integer \( l \) with \( 0 \leq l \leq k \).

### 2.2.3 Eppstein et al.'s sparsification technique

Eppstein et al.'s sparsification technique [44] is based on the sparse certificate concept. Before introducing their technique, we reproduce some of their definitions as follows.

**Definition 2.2** For any graph property \( \mathcal{P} \), and graph \( G(V,E) \), a certificate for \( G \) is a graph \( G' \) such that \( G \) has property \( \mathcal{P} \) if and only if \( G' \) has the property.

**Definition 2.3** For any graph property \( \mathcal{P} \) and a graph \( G \), a strong certificate for \( G \) is a graph \( G' \) on the same vertices set such that, for any \( H \), \( G \cup H \) has the property \( \mathcal{P} \) if and only if \( G' \cup H \) has the property.

**Definition 2.4** Let \( \mathcal{A} \) be a function mapping graphs to strong certificates. Then \( \mathcal{A} \) is stable if it has the following two properties: (1) For any graphs \( G \) and \( H \), \( \mathcal{A}(G \cup H) = \mathcal{A}(\mathcal{A}(G) \cup H) \). (2) For any graph \( G \) and edge \( e \in E \), \( \mathcal{A}(G - \{ e \}) \) differs from \( \mathcal{A}(G) \) by \( O(1) \) edges.

The basic idea behind Eppstein et al.'s sparsification technique is as follows. We maintain a partition of the edges of \( G(V,E) \) into \( \lceil m/n \rceil \) groups, all but one of which contain exactly \( n \) edges. The remaining group which we call the small group may contain between 1 and \( n \) edges. When we insert an edge into \( G \), we place it in the small group. When we delete an edge, we move another edge from the small group into the group.
from which the edge is deleted, to keep the group size invariant (i.e., \( n \) edges exactly). If we delete the small group’s last edge, we remove the group, and if we insert an edge when there are already \( n \) edges in the small group, we start a new small group. After finishing the partition of groups, we establish a complete binary tree \( B^T \) with \( \lceil m/n \rceil \) leaves corresponding to the \( \lceil m/n \rceil \) groups. \( B^T \) is called the sparsification tree in which every node corresponds to a subgraph formed by the edges in the graphs at the leaves of \( B^T \) that are the descendents of the given node. For each such a subgraph, we maintain a sparse certificate \( \mathcal{P} \). The certificate at a given node is found by applying the procedure of finding a sparse certificate to the graph that consists of the union of the certificates of its two children nodes. Each change in the number of groups will cause a single leaf node of \( B^T \) to split, or two leaves to merge. Since we only allow edge insertions and deletions, the number of edges per group will never change. When an edge is inserted or deleted, it only concerns \( O(1) \) groups. For each of these groups, and its \( O(\log \frac{m}{n}) \) ancestor nodes in \( B^T \), we recompute (or dynamically maintain) a new sparse certificate. This results in a sparse certificate of the graph \( G \) at the root of \( B^T \). We update the data structure for property \( \mathcal{P} \) on the graph formed by the sparse certificate at the root of \( B^T \).

2.3 The Maintenance of 2ECCs on Connected Graphs

2.3.1 The data structures

Before we proceed, we introduce some data structures for our algorithm below. Let \( G \) be a connected graph with maximum degree three and \( T \) be a spanning tree of \( G \). Based on the topology of \( T \), partition the vertex set of \( G \) into disjoint subsets such that the subgraph induced on each subset is connected, and each such a subset is a vertex cluster (cluster for short). The size of each cluster is \( O(K) \), where \( K \) will be determined later. In the following we show how to implement the vertex cluster partition in detail. We first generate a spanning tree \( T(V, E_T) \) of \( G \) using an efficient parallel algorithm such as the algorithm by Awerbuch et al. [14], which costs \( O(\log m) \) time with \( O(m+n) \) processors. Then, assume that \( T \) has been built, and \( T \) is represented by an inverted tree. The aim to implement the vertex cluster partition such that the size of each cluster is \( O(K) \), i.e., the size of each cluster is between \( 1/2K \) and \( 3/2K \). Note that the restricted partition method by Frederickson is not applicable because it seems highly sequential. Instead, we must look for a parallel partition approach to implement the vertex cluster partition. Fortunately, Lipton et al. [128] have proven the following lemma which is the cornerstone of our parallel algorithm for the vertex cluster partition.

**Lemma 2.4** [128] Given an \( n \)-vertex tree with maximum degree three, there exists an edge whose removal separates the tree into two subtrees, and both of these two subtrees contain between \( n/3 \) and \( 2n/3 \) vertices.

Using Lemma 2.4, Liang et al. [117] showed there is such a partition by the following lemma.
Lemma 2.5 [117] Given an inverted tree $T$ with maximum degree three, partition its vertex set into $O(m/K)$ clusters such that each cluster has $O(K)$ vertices. This can be done in $O(\log(m/K))$ time using $O(m)$ processors, where $m$ is the number of vertices in $T$, $m \geq 3/2K$.

Let $C$ be a cluster in the vertex cluster partition of $T$. An edge of $G$ with both endpoints in $C$ is said to be internal to $C$, while an edge of $G$ with only one endpoint in $C$ is said to be incident to $C$. The induced subgraph of $G$ by $C$ is denoted by $G(C)$.

The full representation of $C$ is a graph $F(C)$, which consists of $G(C)$ plus the clusters adjacent to $C$ and the edges incident to $C$. In other words, the vertex set of $F(C)$ consists of the vertices internal to $C$ and the clusters incident to $C$. The edge set of $F(C)$ is composed of all the edges of $G$ internal and incident to $C$. Since $C$ contains at most $O(K)$ vertices and each vertex has degree no more than three, $F(C)$ has at most $O(K)$ vertices and edges. It is easily shown that $F(C)$ can be constructed in $O(\log K)$ time using $O(K)$ processors, using the adjacency list of $G(C)$.

The tree representation of $C$ is a graph $T(C)$ which is defined as follows. Its edges are the bridges of $G(C)$ and its vertices are the 2ECCs of $G(C)$. Since $C$ contains at most $O(K)$ vertices, the tree representation of $C$ has size of at most $O(K)$. It is not difficult to show that $T(C)$ is a tree. Unless otherwise specified, in the rest we assume that $T(C)$ is a rooted tree. Furthermore, associated with $T(C)$, there is another data structure — a tree $T''$, which will be used to support the fast queries of any LCA of two vertices of $T(C)$ in $T(C)$, i.e., the LCA queries on $T(C)$ can be answered in $O(1)$ time by calculating on $T''$. Such a $T''$ can be built in $O(\log K)$ time using $O(K)$ processors by the algorithm due to Schieber et al [154].

Lemma 2.6 The full tree representation $T(C)$ of a cluster $C$ can be constructed in $O(\log K)$ time with $O(K)$ processors.

Proof. Since $G(C)$ is a graph with $O(K)$ vertices and with maximum vertex degree three, a spanning tree $T''$ of $G(C)$ can be constructed in $O(\log n)$ time using $O(K)$ processors. Then, find all 2VCCs of $G(C)$ by applying Tarjan et al’s algorithm for biconnectivity [164]. This needs $O(\log K)$ time using $O(K)$ processors. Note that the edge in a 2VCC consisting of two vertices is a bridge of $G(C)$. Denote by $B$ the set of all bridges of $G(C)$. Let $G'(C)$ be the graph obtained by deleting all the edges in $B$ from $G(C)$. Obviously, $G'(C)$ can be constructed in $O(1)$ time with $O(K)$ processors because of the constant degree of each vertex in $G(C)$. Having $G'(C)$, find the CCs of $G'(C)$ which are the 2ECCs of $G(C)$. This can be done in $O(\log K)$ time with $O(K)$ processors [159]. Label the vertices in a 2ECC by a unique identification. For convenience, we use the smallest vertex index in a 2ECC to label its vertices and to represent the 2ECC. The details are as follows. Let $D()$ be a labeling function, and $BC$ be a 2ECC. Then, $D(u) := v$ for all $u \in BC$, where $v$ is the vertex with the smallest index in $BC$. Having done above, we now consider the construction of $T(C)$ as follows. The vertex set of $T(C)$ consists of all 2ECCs of $G(C)$. Let $(u,v)$ be an edge of $T(C)$ and $u$ be the parent of $v$ in $T(C)$. In the following we will use the directed edge $\langle v,F_{T(C)}(v) \rangle$ to represent the edge $(u,v)$, where $F_{T(C)}(v) = u$. Then, the edge set...
of $\mathcal{T}(C)$ is obtained as follows. Initially, $F_{T[C]}(D(v)) := v$ for each $v$ of $\mathcal{G}(C)$. Then, for every edge $(x, y)$ in $B$, $F_{T[C]}(D(x)) := D(y)$ if $D(y)$ is the parent of $D(x)$ in $T''$, otherwise $F_{T[C]}(D(y)) := D(x)$. This can be done in $O(1)$ time with $O(K)$ processors because there are $O(K)$ edges in $B$. \qed

Let $C_i$ and $C_j$ be two clusters, and $E_{i,j} = \{e_1, e_2, \ldots, e_h\}$ be the edge set of $G$ between $C_i$ and $C_j$. The edges in $E_{i,j}$ is represented by two data structures: one for $C_i$, denoted by $E(C_i, C_j)$, and the other for $C_j$, denoted by $E(C_j, C_i)$. $E(C_i, C_j)$ stores the information about the endpoints of the edges in $E_{i,j}$ which belong to $C_j$. In other words, a vertex $u \in \mathcal{T}(C_j)$ is included in $E(C_i, C_j)$ if there is an edge in $E_{i,j}$ incident to a vertex of $u$. For example, let $R$ be a vertex of $\mathcal{T}(C_j)$, a vertex $x \in V$ be included in $R$, and $(x, y) \in E_{i,j}$. Then, the vertex $R$ is included in $E(C_i, C_j)$ by the definition. $E(C_i, C_j)$ is represented by a balanced binary tree and the key to access this tree is the pre-order numbering of vertices in $\mathcal{T}(C_j)$. All $E(C_j, C_i)$ for a cluster $C_i$ can be established by sorting in $O(\log K)$ time using a total of $O(K)$ processors, because the degree of each vertex in $G$ is no more than three and the number of vertices in $C_i$ is $O(K)$, where $C_j$ is an adjacent cluster of $C_i$.

The supergraph $\tilde{G}$ of $G$ is a multigraph in which each vertex represents a cluster. Let $C_i$ and $C_j$ be two clusters. There is an edge between $C_i$ and $C_j$ if $|E_{i,j}| = 1$; otherwise there are two edges between them if $|E_{i,j}| \geq 2$.

An equivalent graph $\overline{\tilde{G}}(V, E)$ of $\tilde{G}$ is defined as follows. $V = \{C_i \mid C_i \text{ is a cluster}\} \cup \{v_{i,j,1}, v_{i,j,2} \mid PP1\}$, where $v_{i,j,k}$ is a new added vertex, $k = 1, 2$, and $PP1$ denotes that there are two edges between $C_i$ and $C_j$ in $\tilde{G}$, and $E = \{(C_i, C_j) \mid PP2\} \cup \{(C_i, v_{i,j,k}), (v_{i,j,k}, C_j) \mid PP3 \text{ and } k = 1, 2\}$, where $PP2$ denotes that there is only one edge in $\tilde{G}$ between $C_i$ and $C_j$, and $PP3$ denotes that there are two edges between $C_i$ and $C_j$ in $\tilde{G}$. Obviously, $\overline{\tilde{G}}$ is a simple graph. We say that both $\tilde{G}$ and $\overline{\tilde{G}}$ are “equivalent”, which means that both of them have the same sets of articulation points and bridges.

From now on, assume that there is no difference between $\tilde{G}$ and $\overline{\tilde{G}}$ under the equivalence sense. Clearly if an adjacent vertex $v$ of a vertex $C$ in $\overline{\tilde{G}}$ is not a vertex of $\tilde{G}$, there must exist a unique corresponding vertex $C'$ in $\tilde{G}$ such that there are two edges between $C'$ and $C$ in $\tilde{G}$. In the following we say that the adjacent clusters of a cluster means the adjacent clusters of this cluster in $\tilde{G}$.

**Lemma 2.7** Given a connected graph with $n_1$ vertices and $m_1$ edges, deciding if there exist bridges on any path between two vertices $x$ and $y$ can be done in $O(\log n_1)$ time with $O(n_1 + m_1)$ processors.

**Proof.** Omitted. \qed

Let $\mathcal{G}_{x,y}$ be a graph by replacing the vertices $C_x$ and $C_y$ in $\overline{\tilde{G}}$ with their full representations. Then, the construction of $\mathcal{G}_{x,y}$ can be carried out in $O(\log K + \log(m/K))$ time using $O(K + (m/K)^2)$ processors by merging the adjacency lists of $\mathcal{G}(C_x)$, $\mathcal{G}(C_y)$, and $\overline{\tilde{G}}$. 
Corollary 2.1 Deciding if there is a bridge between \( x \) and \( y \) in \( G_{xy} \) can be done in \( O(\log(K + m/K)) \) time using \( O(K + (m/K)^2) \) processors. The maximum degree of \( G(C_x) \) or \( G(C_y) \) is three.

Proof. Its proof is easy by Lemma 2.6, because \( G_{xy} \) has \( O(K + m/K) \) vertices and \( O(K + (m/K)^2) \) edges at most. \( \square \)

2.3.2 An algorithm for queries

Having built the above data structures, we now deal with the query issue. Let \( x \) and \( y \) be two vertices that are not in a 2ECC of \( G \). Then there is a bridge separating \( x \) and \( y \) in \( G \). Any such a bridge \( e \) can be of two different types: type (1) if \( e \) is either internal to \( C_x \) and \( C_y \) or an edge between two different clusters; and type (2) if \( e \) is internal to a cluster \( C, C \neq C_x \) and \( C \neq C_y \).

Theorem 2.2 [66] Let \( x \) and \( y \) be any two vertices of \( G \). Then the following is true:

(i) \( e \) is a type (1) bridge in \( G \) if and only if the corresponding edge is a bridge in \( G_{xy} \);

(ii) If \( e \) is a type (2) bridge separating \( x \) and \( y \) in \( G \), and \( e \) is internal to a cluster \( C (C \neq C_x \) and \( C \neq C_y) \), then \( C \) is an articulation point of \( G_{xy} \).

Let us recall Galil et al's algorithm for a query. Their algorithm is basically divided into two phases. In the first phase it checks type (1) bridges. If there are no type (1) bridges, then it enters the second phase by checking type (2) bridges. The type (1) bridge can be easily detected by Corollary 2.1. The crucial task therefore is how to check efficiently whether there is a bridge separating \( x \) and \( y \) within a cluster \( C \), where \( C \) is an articulation point of \( G_{xy} \) between \( x \) and \( y \). Obviously it is easy to show that \( C \) is an articulation point of \( \overline{G} \) between \( C_x \) and \( C_y \) too.

Lemma 2.8 All articulation points of \( \overline{G} \) which separate \( C_x \) from \( C_y \) can be found in \( O(\log(m/K)) \) time using \( O((m/K)^3) \) processors.

Proof. First, make a copy of \( \overline{G} \) for each cluster \( C \) in \( \overline{G} \). Then, decide whether \( C \) is an articulation point of \( \overline{G} \) separating \( C_x \) from \( C_y \) in parallel, which proceeds as follows: delete \( C \) from \( \overline{G} \), compute all CCs of the remaining graph \( \overline{G} - C \). If \( C_x \) and \( C_y \) are not in the same CC of \( \overline{G} - C \), then \( C \) is an articulation point of \( \overline{G} \) that separates \( C_x \) from \( C_y \). Since the computation of CCs for each \( C \) requires \( O(\log(m/K)) \) time and \( O((m/K)^2) \) processors [159], the computation of all articulation points of \( \overline{G} \) between \( C_x \) and \( C_y \) requires \( O(\log(m/K)) \) time with \( O((m/K)^3) \) processors. \( \square \)

In order to present Galil et al's algorithm for checking type (2) bridges, we introduce some concepts first. Let \( C \) be an articulation point in a path between \( C_x \) and \( C_y \) in \( \overline{G} \), which is also an articulation point of \( G_{xy} \), and let \( AC = \{ Q_1, Q_2, \ldots, Q_t \} \), where \( Q_j \) is an adjacent cluster of \( C \) in \( \overline{G} \), \( 1 \leq j \leq t \) and \( t \geq 2 \). Assume that there are \( s \) CCs in the remaining graph after deleting \( C \) from \( \overline{G} \), and they are \( W_1, W_2, \ldots, W_s \). Assume that \( C_x \) is in \( W_1 \) and \( C_y \) is in \( W_2 \). Let the clusters of \( AC \) in \( W_1 \) be \( X_1, X_2, \ldots, X_p \), the
clusters of $AC$ in $W_2$ be $\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_q$, and the other remaining clusters of $AC$ in $W_i$ be $Z_{i,1}, Z_{i,2}, \ldots, Z_{i,r(i)}$. Then $p + q + \sum_{i=3}^{t} r(i) = t$.

Let $A_1, A_2, \ldots, A_l$ be any $l$ clusters adjacent to cluster $C$, $l > 0$. Then, all $E(A_i, C) \neq \emptyset$, $1 \leq i \leq l$. Color all vertices in $E(A_i, C)$ with color blue, $1 \leq i \leq l$. Let $SV$ be the set of blue vertices in $T(C)$. Denote by $\lambda(A_1, A_2, \ldots, A_l)$ the lowest common ancestor (LCA) of $SV$ in $T(C)$. The LCA of a vertex set $U$ is defined as follows: (i) the LCA of $U$ is $u$ if $U = \{u\}$; (ii) the LCA of $U$ is equal to the LCA of $(U - \{u, v\}) \cup \{LCA(u, v)\}$ for any $u \in U$ and $v \in U$ otherwise. The computation of LCA of $U$ can be obtained by sorting the vertices by their pre-order numbering in $T(C)$. Let the sorted vertex sequence be $v_1, v_2, \ldots, v_k$, then it is easy to prove that the lowest common ancestor of $U$ is $LCA(v_1, v_k)$.

From the definition above, $\lambda(A_1, A_2, \ldots, A_l)$ is such a vertex in $T(C)$ that all the edges between $A_i$ and $C$ are incident to below $\lambda(A_1, A_2, \ldots, A_l)$, $1 \leq i \leq l$. Here “below” means the other endpoints of these edges in $T(C)$ fall into the subtree rooted at $\lambda(A_1, A_2, \ldots, A_l)$.

A vertex $v$ of $T(C)$ is colored red if there is an edge of $G$ connected a vertex in $\mathcal{X}_i$ and a vertex in $v$ for some $i$, $1 \leq i \leq p$. Similarly, a vertex $v$ of $T(C)$ is colored black if there is an edge of $G$ connected a vertex in $\mathcal{Y}_j$ and a vertex in $v$ for some $j$, $1 \leq j \leq q$. Define the top-red vertex $\rho$ as $\lambda(\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_p)$. Notice that $\rho$ is the LCA of all the red vertices of $T(C)$. Similarly, define the top-black vertex $\beta$ as $\lambda(\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_q)$.

**Lemma 2.9** [66] Let $\rho$ be the top-red vertex and $\beta$ the top-black vertex, and let $v = LCA(\beta, \rho)$. Let $\pi_{\rho, \beta}$ be the path in $T(C)$ between $\rho$ and $\beta$. Then

1. If $v = \rho = \beta$, no edge in $T(C)$ can separate black and red vertices;
2. If $v \neq \rho$ and $v \neq \beta$, then all the edges in $\pi_{\rho, \beta}$ separate black and red vertices;
3. If $v = \beta \neq \rho$, an edge $e$ separates black and red vertices if and only if $e$ is in $\pi_{\rho, \beta}$ and there are no red vertices below $e$;
4. If $v = \rho \neq \beta$, an edge $e$ separates black and red vertices if and only if $e$ is in $\pi_{\rho, \beta}$ and there are no black vertices below $e$.

**Lemma 2.10** $\lambda(A_1, A_2, \ldots, A_l)$ can be found in $O(\log l + \log K)$ time with $O(l)$ processors.

**Proof.** First, find two vertices $u_i$ and $v_i$ from $E(A_i, C)$ for every $A_i$ such that $u_i$ has the smallest pre-order numbering in $T(C)$ and $v_i$ has the largest pre-order numbering in $T(C)$, which can be found in $O(\log K)$ time with $O(1)$ processors. Then, find $\alpha_i = LCA(u_i, v_i)$ for each $A_i$ in $T(C)$, which can be done in $O(1)$ time with $O(1)$ processors by calculating on the tree $T_i$ that is associated with $T(C)$, $1 \leq i \leq l$. Finally, sort the sequence $\alpha_1, \alpha_2, \ldots, \alpha_l$ by their pre-order numberings, which can be done in $O(\log l)$ time with $O(l)$ processors. $\lambda(A_1, A_2, \ldots, A_l)$ is the LCA of the two vertices with the smallest and the largest pre-order numberings in the sorted sequence. So, the entire computation takes $O(\log l + \log K)$ time and uses $O(l)$ processors. □

**Corollary 2.2** The computation of $\rho$ and $\beta$ can be carried out in $O(\log(m/K) + \log K)$ time using $O(m/K)$ processors.
Proof. It can be derived from Lemma 2.10, because $1 \leq p, q \leq cm/K$ and $c$ is a constant. □

Lemma 2.11 [66] Let $C'$ be any cluster adjacent to $C$, and let $v$ be any vertex of $T(C)$. Then we can check whether there are edges between $C'$ and $C$ incident to below $v$ in $O(\log K)$ time.

Lemma 2.9 actually gives an algorithm for checking whether there is a bridge in $\pi_{p,\beta}$ of $T(C)$ after considering the edges of $G$ incident to vertices of the clusters in $\mathcal{W}_1$ and $\mathcal{W}_2$. If there is a bridge $e$ in $\pi_{p,\beta}$ which separates red vertices from black vertices, then all the other edges of $G$ between $C$ and the clusters in $\mathcal{W}_3, \mathcal{W}_4, \ldots, \mathcal{W}_s$ are needed to be checked, in order to make sure whether or not $e$ is really a bridge of $G$. How to check this part (Case (3) of Lemma 2.9) efficiently is the key part of Galil et al’s algorithm. However, their algorithm for dealing with this part is highly sequential, and therefore is hard to parallelize. In the following an efficient parallel algorithm that uses a different approach to accomplish that is suggested.

Consider $\mathcal{W}_3, \mathcal{W}_4, \ldots, \mathcal{W}_s$. Let $\gamma_i = \lambda(Z_{i,1}, Z_{i,2}, \ldots, Z_{i,r(i)})$ where $Z_{i,j} \in \mathcal{W}_i$ is a cluster and $1 \leq j \leq r(i)$. If some $\gamma_i$ is not in the path of $T(C)$ between $\rho$ and the root of $T(C)$, then the removal of any edge in $\pi_{p,\beta}$ leaves all vertices in $E(Z_{i,j}, C)$ on the same side, $1 \leq j \leq r(i)$, which means that the edges between $C$ and the clusters in $\mathcal{W}_i$ do not help to remove the bridges in $\pi_{p,\beta}$, $3 \leq i \leq s$. So, we first examine all $\gamma_i$ by deleting those vertices which are not in the path of $T(C)$ between $\rho$ and the root of $T(C)$. This can be implemented in $O(1)$ time using $O(s)$ processors by testing if $\gamma_i = LCA(\rho, \gamma_i), s \leq cm/K$. We then sort the remaining vertices by their pre-order numberings. Let $\gamma'_0, \gamma'_1, \gamma'_2, \ldots, \gamma'_{s'}$ be the resulting sequence, and $\mathcal{W}'_0, \mathcal{W}'_1, \ldots, \mathcal{W}'_{s'}$ be the corresponding CCs, where $0 \leq s' \leq s$ and $\gamma'_0 = \rho$. Note that all $\gamma'_i$ are in $\pi_{p,\gamma'_i}$ and $pre(\gamma'_i) \geq pre(\gamma'_{i+1})$, where $pre(r)$ is the pre-order numbering of vertex $r$ in $T(C)$, $0 \leq i \leq s'$.

Our approach to dealing with Case (3) of Lemma 2.9 is to construct the following auxiliary graphs $G_i$, first, $i = 1, 2$, and then to check whether there is a bridge in $\pi_{p,\beta}$ with the information supplied by the auxiliary graphs. The algorithm proceeds as follows. First, label all clusters in $\mathcal{W}'_i$ with $\gamma'_i$, $0 \leq i \leq s'$. Then, an auxiliary undirected graph $G_1(V_1, E_1)$ is constructed, where $V_1 = \{\gamma'_0, \gamma'_1, \ldots, \gamma'_{s'}\}$, and an edge $(\gamma'_i, \gamma'_j) \in E_1$ if either (i) $E_{i,j} \neq \emptyset$, $\gamma'_i \in \mathcal{W}'_i$, and $\gamma'_j \in \mathcal{W}'_j$; or (ii) $\gamma'_i \in \mathcal{W}'_i$, and there is an edge between $C$ and $\mathcal{C}_j$ incident to below $\gamma'_i$ in $T(C)$. It is not difficult to verify that $G_1$ can be constructed in $O(\log K)$ time using $O((m/K)^2)$ processors because $1 \leq s' \leq cm/K$. Third, compute the transitive closure $G^*$ of $G_1(V_1, E_1)$, which can be obtained by computing all CCs of $G_1$ because $G_1$ is an undirected graph. Finding all CCs of $G_1$ requires $O(\log (m/K))$ time and $O((m/K)^2)$ processors because $G_1$ contains $O(m/K)$ vertices [159]. Now for each CC of $G_1$, we find the largest and the smallest indexes of the vertices in it, and broadcast them to all vertices in it. This requires $O(\log (m/K))$ time and $O(m/K)$ processors. After that, each vertex $v$ in a CC receives a label pair, $l$ and $h$, where $l$ is the smallest index of vertices and $h$ is the largest index of vertices in the CC containing $v$. Let $[l, h]$ represent a closed integral interval, and $l \leq h$. Denote by $[l_i, h_i]$ the label pair the vertex $\gamma'_i$ received.
Another undirected graph $G_2(V_2, E_2)$ is constructed as follows: $V_2 = V_1$ and $(\gamma'_i, \gamma'_j) \in E_2$ if either $(\gamma'_i, \gamma'_j) \in E_1$ or $[l_i, h_i] \cap [l_j, h_j] \neq \emptyset$. Then, we have

**Lemma 2.12** Let $CC_{\rho}$ be a CC of $G_2(V_2, E_2)$ containing vertex $\rho = \gamma'_0$ such that $i_0$ is the maximum index of all vertices in $CC_{\rho}$. Then there is no bridge in $\pi_{\rho, \gamma'_0}$, where $\pi_{\rho, \gamma'_0}$ is a sub-path of $\pi_{\rho, \beta}$ of $T(C)$.

**Proof.** Let $CC'$ be a CC of $G_1$ consisting of vertices $\gamma'_{i_1}, \gamma'_{i_2}, \ldots, \gamma'_{i_k}$. Assume that these vertices are sorted already by their pre-order numberings, i.e., $\text{pre}(\gamma'_{i_j}) \geq \text{pre}(\gamma'_{i_{j+1}}), 1 \leq j \leq k-1$. Then, there is no bridge in the maximal subpath $\pi_{\gamma'_{i_1}, \gamma'_{i_k}}$ of $\pi_{\rho, \beta}$ in $T(C)$ because each edge in this subpath is included in a cycle at least (see Figure 2.1), and $\pi_{\gamma'_{i_1}, \gamma'_{i_k}}$ can be found by computing the transitive closure of $G_1$, i.e., computing the CCs of $G_1$. Let $CC_i$ and $CC_j$ be two CCs of $G_1$, and $[l_i, h_i]$ and $[l_j, h_j]$ be the corresponding label pairs. If $[l_i, h_i] \cap [l_j, h_j] \neq \emptyset$, there are the following four cases: (i) $l_i > l_j$ but $h_i < h_j$, i.e., $[l_i, h_i] \subset [l_j, h_j]$, the path $\pi_{\gamma'_{i_k}, \gamma'_{h_k}}$ is a subpath of $\pi_{\gamma'_{i_k}, \gamma'_{h_k}}$ in $T(C)$, and all edges in $\pi_{\gamma'_{i_k}, \gamma'_{h_k}}$ are not bridges; (ii) $l_i < l_j$ and $h_i < h_j$, there is no any bridges in $\pi_{\gamma'_{i_k}, \gamma'_{h_k}}$ in $T(C)$. Notice that both $\pi_{\gamma'_{i_k}, \gamma'_{h_k}}$ and $\pi_{\gamma'_{i_k}, \gamma'_{h_k}}$ are the subpaths of $\pi_{\gamma'_{i_k}, \gamma'_{h_k}}$ in $T(C)$, and they can be found by constructing $G_2$ and computing its transitive closure; (iii) $l_i < l_j$ but $h_i > h_j$, the discussion is similar to case (i), omitted; (iv) $l_i > l_j$ and
\[ h_i > h_j, \] the discussion is similar to case (ii). By the discussions above, we conclude there is no any bridges in \( \pi_{\rho, \gamma_{i_0}} \) of \( T(C) \). \( \square \)

We now present the NC algorithm for deciding whether there is a bridge in \( \pi_{\rho, \beta} \) of \( C \) as follows. If \( \text{pre}(\beta) > \text{pre}(\alpha_{i_0}) \), then \( x \) and \( y \) are in the same 2ECC of \( G \). Otherwise, we check whether there is a bridge in \( \pi_{\alpha_{i_0}, \beta} \). This can be done by checking whether there is a cluster \( Y_j \in \mathcal{W}_2 \) such that \( u \in E(Y_j, C) \) and \( \text{pre}(\alpha_{i_0}) \leq \text{pre}(u) < \text{pre}(\alpha_{i_0}^l) + \text{nd}(\alpha_{i_0}^l) \), where \( \text{nd}(z) \) is the number of vertices in the subtree of \( T(C) \) rooted at \( z \). In other words, we need to check whether there is such an edge \((u', v')\) of \( G \) that \( u' \in A \), \( v' \in Y_j \), and the vertex \( A \) is included in the subtree of \( T(C) \) rooted at \( \alpha_{i_0}^l \), \( 1 \leq j \leq q \). The checking above can be done in \( O(\log K) \) time using \( O(m/K) \) processors by Lemma 2.10, because \( q \leq cm/K \). If this is such a \( Y_j \), there is no bridge in \( C \), we check the other articulation points of \( G \) separating \( C_x \) and \( C_y \) in \( G \). Otherwise, \( x \) and \( y \) of \( G \) are not in the same 2ECC of \( G \). So, the checking of Case (3) in Lemma 2.9 (checking if there is a bridge in all articulation points of \( G \) which separates \( C_x \) and \( C_y \) in it) can be done in \( O(\log (m/K) + \log K) \) time using \( O((m/K)^3) \) processors since there are at most \( O(m/K) \) articulation points of \( G \). Case (4) in Lemma 2.9 is similar to Case (3). Case (2) can be divided into two substeps, by checking whether there exist bridges in \( \pi_{\rho, v} \) and \( \pi_{\beta, v} \) respectively. This kind of checking can be done using the approaches for Case (3) and Case (4). We thus have the following theorem.

**Theorem 2.3** The query about whether \( x \) and \( y \) are in the same 2ECC of a connected graph \( G \) can be answered in \( O(\log m) \) time using \( O(m^{3/4}) \) processors.

**Proof.** It is straightforward from the previous theorems and lemmas by setting \( K = m^{3/4} \). \( \square \)

### 2.3.3 Inserting and deleting edges

We now show how to implement InsertEdge \((x, y)\) operation. When a new edge \((x, y)\) is inserted, the degrees of vertices \( x \) and \( y \) in the original graph are increased by one. If either one of them becomes four, then the transformation in Section 2.2 will be applied to the vertex with degree four again. In both cases, this introduces at most a constant number of extra vertices and edges which can be dealt with easily. However the insertion of extra vertices to keep the vertex degree no greater three may lead to the size of either \( C_x \) or \( C_y \) becoming greater than \( 3/2K \). We deal with this latter case by splitting \( C \) into two new clusters \( C' \) and \( C'' \). Since the size of \( C \) is \( O(K) \) and the degree of vertices in \( C \) is no greater than three, the two clusters \( C' \) and \( C'' \) and the data structures related to them can be established in \( O(\log K) \) time using \( O(K) \) processors. In detail, the full representations, \( F(C') \) and \( F(C'') \) of \( C' \) and \( C'' \), and their tree representations \( T(C') \) and \( T(C'') \) can be calculated in \( O(\log K) \) time with \( O(K) \) processors by the discussion in Section 2.3.1. The balanced binary trees for all other cluster \( C_j \), \( E(C', C_j) \), \( E(C_j, C') \), \( E(C'', C_j) \) and \( E(C_j, C'') \) can be constructed in \( O(\log K) \) time with \( O(K) \) processors because the degree of each vertex in \( C \) is at most three,
where \( \mathcal{C}_j \) is an adjacent cluster of \( \mathcal{C} \). The graph \( \overline{G} \) can be updated in \( O(\log(m/K)) \) time using \( O((m/K)^2) \) processors.

In summary, inserting an edge \((x, y)\) causes at most two clusters to be split and a constant number of clusters to be updated, and each update requires \( O(\log K + \log(m/K)) \) time with \( O(K + m/K)^2 \) processors. In order to keep \( O(\log m) \) time complexity per update and per query in the worst case, we need to be cautious when dealing with the data structures above. Let \( m_t \) be the number of edges in the graph at time \( t \). We show that an update at time \( t \) can be implemented in \( O(m_t^{3/4}) \) time with \( O(m_t^{3/4}) \) processors. Let \( K_t = \lceil m_t^{3/4} \rceil \). When the value of \( K \) changes due to an \textit{InsertEdge} or \textit{DeleteEdge} operation, there will be at least \( 1/2\lceil m_t^{3/4} \rceil \) more updates before \( K \) becomes twice as large or one half as small as it was before. Following the same idea as Galil et al. [66], we adjust a constant number of clusters each time if there is such an update. This gives a total of \( O(m_t^{3/4}) \) cluster adjustments. Since there are no more than \( O(m_t/K_t) = O(m_t^{1/3}) \) clusters that needs to be adjusted, the adjustments may be accomplished before a new round of adjustments issues. Therefore, when an insertion happens, scan the related clusters, and find any cluster that is too small, and merge a constant number of such clusters with their neighboring clusters if needed. The processing for the deletion case is similar to that for the insertion case, so we omit it here. Therefore, we have

**Theorem 2.4** Full dynamic maintenance of 2ECCs of a connected graph can be done in \( O(\log n) \) time using \( O(m^{3/4}) \) processors on a PRIORITY CRCW PRAM.

### 2.4 The Maintenance of 2- and 3-ECCs on General Graphs

**2.4.1 2ECCs on disconnected graphs**

In this section we discuss the fully dynamic maintenance of 2ECCs on disconnected graphs by extending the results on connected graphs in the previous section. Let \( G \) be a disconnected graph. If \( G \) consists of \( l \) CCs, we augment \( G \) by adding \( l - 1 \) dummy edges such that the augmented graph is connected and there is no cycle that contains two dummy edges. This does not change the property of 2ECCs of \( G \), i.e., two vertices \( x \) and \( y \) are in a 2ECC of the augmented graph if and only if they are in the same 2ECC of \( G \). Furthermore, we assign the edges in \( G \) with cost 1 and the dummy edges with cost 2, and maintain a topological partition of an MST \( T \) of the augmented graph. When inserting an edge \( e = (x, y) \), we check whether there is already a dummy edge between \( x \) and \( y \) in \( T \). If so, we decrease its cost from 2 to 1. Otherwise we insert this edge with cost 1 into \( T \), and update \( T \) by applying Liang et al.'s algorithm [117] which can be implemented in \( O(\log m) \) time with \( O(m^{2/3}) \) processors. If \( e \) becomes an edge in the new MST by deleting a dummy edge \( e' \), we just delete \( e' \).

When deleting an edge \( e = (x, y) \), we first check whether \( e \) is in \( T \). If \( e \) is not in \( T \), we simply delete \( e \), otherwise we update \( T \). Therefore, we have the following theorem.

**Theorem 2.5** Fully dynamic maintenance of 2ECCs of a disconnected graph \( G \) can be done in \( O(\log n) \) time using \( O(m^{3/4}) \) processors on a PRIORITY CRCW PRAM.
2.4.2 An improved algorithm for 2ECCs

The proposed algorithm above can be further improved to $O(\log n \log(m/n))$ time using $O(n^{3/4})$ processors only after proper preprocessing if Eppstein et al’s sparsification technique is employed. In the following we describe this improved algorithm.

If $G$ is $k$-edge connected, we can use a sparse $k$-edge certificate $U'_k$ of $G$, defined in Section 2.2 instead of $G$ itself to calculate the $k$-edge connectivity of $G$. Eppstein et al. [44] further proved that $U'_k$ is a strong certificate of $k$-edge connectivity of $G$. Obviously the 2-edge connectivity of $G$ is a special case with $k = 2$.

For any fixed $k$ we maintain the graph $U'_k$ in time $O(\log n)$ with $O(n^{2/3})$ processors per update, using $k$ copies of the NC algorithm for the fully dynamic maintenance of MSFs. Copy $i$ is used to maintain the forest $T'_i$ defined as before, $i = 1, \ldots, k$. The resulting certificate is stable since each update involves at most a single edge flip in each of these $k$ MSFs.

Theorem 2.6 Fully dynamic maintenance of 2ECCs of $G$ can be done in $O(\log n \log \frac{m}{n})$ time using $O(n^{3/4})$ processors on a PRIORITY CRCW PRAM.

Proof. By applying the sparsification technique described in Section 2.2 to each node in the path from a leaf node to the root of $BT$, i.e., maintain the sparse 2-edge certificate at the node through two dynamic data structures, one for 1-edge connectivity which requires $O(\log n)$ time and $O(n^{2/3})$ processors [117], and another for 2-edge connectivity, i.e, $T_1$ and $T_2$. We then run the NC algorithm described in Section 2.3 at the root of $BT$. So, for every edge insertion and deletion, the update time for all data structures is $O(\log n \log \frac{m}{n})$ and the number of processors used is at most $O(n^{3/4})$. The query can be done at the root of $BT$ in the same time and processor bounds as above. □

2.4.3 An algorithm for 3ECCs

Theorem 2.7 Fully dynamic maintenance of 3ECCs of $G$ requires $O(\log n \log \frac{m}{n} + \log n \log \log n/\alpha(3n, n))$ time using $O(n \alpha(3n, n)/\log n)$ processors per update and $O(1)$ time with a single processor per query on a PRIORITY CRCW PRAM.

Proof. Galil et al. [65] give a reduction technique which reduces the $k$-edge connectivity of $G$ into the $k$-vertex connectivity of another graph $G''$. Let $G$ be a sparse 3-edge certificate of the original graph. Obviously, the transformation of $G$ into $G''$ can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors for $k = 3$. By applying Eppstein et al’s sparsification technique and the NC algorithm for the maintenance of 3VCCs [61], the theorem follows. □

Remarks. Galil et al. [64] claimed that their technique for fully dynamic maintenance of 2ECCs can be used for the fully dynamic maintenance of 3ECCs without many changes, and proposed an $O(m^{2/3})$ time sequential algorithm both per update and per query for this latter problem. Their statement leads us to believe that we can derive a
better parallel algorithm with $O(\log n \log \frac{m}{n})$ time and $O(n^{3/4})$ processors for the fully
dynamic maintenance of 3ECCs using the similar technique presented in this section.

2.5 The Maintenance of 2- and 3-VCCs on General Graphs

2.5.1 Finding a strong, sparse $k$-vertex certificate of $G$

Eppstein et al’s sparsification technique is applicable only if the certificate generated
is both sparse and strong. For $k$-vertex connectivity of a graph $G$, Eppstein et al. [44]
show that $Q_k'$ is a strong sparse certificate of $G$, where $Q_k' = \bigcup_{i=1}^{k} B_i$ and $B_i$ is a
breadth-first search forest of $G - \bigcup_{i=1}^{k-1} B_i$. Whereas currently the number of processors used in any NC algorithm for finding a breadth-first search forest is no better
than the number of operations for computing a boolean matrix multiplication which is at least $O(n^{2.376})$ [32]. So, here we adopt an alternative approach to find a sparse
$k$-vertex certificate firstly, and secondly show this certificate is a strong certificate. Finally applying Eppstein et al’s sparsification technique, we obtain an NC algorithm for the fully dynamic maintenance of 2VCCs and 3VCCs of $G$. It should be mentioned
that the technique used to cope with the fully dynamic maintenance of 2VCCs by
Ranuch [147, 148] seems hard to parallelize efficiently. We make this comment based on
the following two reasons. One is that her algorithms are based on a special restricted
decomposition of graph which seems highly sequential. Another is inconsistency be-
tween the data structures used to maintain high level graphs and the data structures
used for shared vertices which makes her algorithms hard to parallelize. Besides that,
the time complexity of her algorithms is amortized. This means that after undergoing
$k$ updates, all data structures must be rebuilt, the amount of work for this rebuilt is
$O(m)$. In the parallel context, we expect to assign at most $o(m)$ work per update
because we have only $o(m)$ processors available.

We start by showing how to produce a sparse $k$-vertex certificate. We then show
that the certificate obtained is strong also. We use the scan-first search technique [21]
to generate a sparse $k$-vertex certificate of graph $G$. Let $C_k$ be a sparse $k$-vertex
certificate of $G$ defined in Section 2.2. It is obvious that $C_k$ is sparse if $k$ is fixed.
In the following we show that $C_k$ is strong also.

Lemma 2.13 $C_k$ is a strong certificate of $k$-vertex connectivity of $G(V,E)$.

Proof. Consider a $(k-1)$-vertex cut $S$ in any graph $C_k \cup H$, $S$ partitions the vertices
in $V - S$ into two subsets $A$ and $B$. Obviously $H$ contains no edge between $A$ and $B$.
Let $s_i \in S - \{s_1, s_2, \ldots, s_{k-1}\}$ be the first vertex in this set of the $i$th scan-first search
of $G_i = G - F_1 - F_2 - \ldots - F_{i-1}$, where $F_i$ is a scan-first search spanning forest of $G_i$.

Let $T$ be a tree containing $s_1$ in $F_1$ and $r$ be the root of $T$. Then either $r \in A$ or
$r \in B$, or $r \in S$. If $r \in S$, we claim that $r = s_1$ by the following argument: the index
of $r$ is smaller than the index of $s_1$ because $r$ is visited before $s_1$ by the definition of
scan-first search. This contradicts the assumption that $s_1$ is the smallest index in $S$.
So, either $r \in A$ or $r \in B$. Without loss of generality, we assume that $r \in A$. We show
that all edges in $G$ that connect $s_1$ and the vertices in $B$ are included in $T$, therefore are included in $F_1$ because $T \in F_1$. We proceed as follows.

If $r = s_1$, all edges incident to $s_1$ are in $T$ obviously. Now consider $r \in A$. Assume that there exists an edge $(s_1, v) \in E$ and $v \in B$ such that $(s_1, v) \notin T$. We show this is impossible. Since $s_1$ and $v$ are in the same CC of $G$, they are in the same spanning tree of $G$. Therefore both $s_1$ and $v$ are included in $T$. Let $p(v)$ be the parent of $v$ in $T$. Then either $p(v) \in A$ or $p(v) \in B$, or $p(v) \in S$. If $p(v) \in A$, then, after deletion all vertices in $S$ and the edges incident to these vertices from $G$, there exists an edge connecting $p(v)$ ($\in A$) and $v$ ($\in B$) which contradicts that $S$ is a $(k - 1)$-vertex cut between $A$ and $B$. Thus, either $p(v) \in B$ or $p(v) \in S$. Obviously it is also impossible that $p(v) \in B$ because the index of $s_1$ by scan-first is smaller than that of any vertex in $B$. So, $s_1$ should be selected as the parent of $v$ in $T$ because $(s_1, v) \in E$. By the discussion above, we conclude that $p(v)$ must be in $S$ and $p(v) = s_1$, i.e., the edge $(s_1, v)$ must be included in $T$, therefore must be included in $F_1$. In other words, all edges connecting $s_1$ and the vertices in $B$ are included in $F_1$ instead of $C_k - F_1$, and $S - \{s_1\}$ forms a $(k - 2)$-vertex cut in $C_k - F_1$. We proceed on the remaining graph in the same way. As a result, each application of scan-first search will eliminate a vertex from $G$, and in the end $F_k$ must be disconnected. But this only happens if $S$ is also a vertex cut set of $G$, and hence of $G \cup H$. Therefore we have already shown that any vertex cut in $C_k \cup H$ is also a vertex cut in $G \cup H$. The converse follows immediately from the fact that $C_k$ is a subgraph of $G$. Hence $C_k$ is a strong certificate of $G$. □

2.5.2 The maintenance of 2VCCs

Assume that $G$ is connected. If not, we maintain two types of data structures, one for 1-edge connectivity and another for biconnectivity. That is, if there is a query to ask whether vertices $u$ and $v$ are in a 2VCC, we first check the data structures for 1-edge connectivity to see whether $u$ and $v$ are connected. If yes, we run the procedure of query on the data structures for biconnectivity. Otherwise, the response to the query is false. It is known that the fully dynamic maintenance of 1-edge connectivity can be done in $O(\log n \log \frac{m}{\sqrt{n}})$ time using $O(n^{2/3})$ processors on the same model [117].

**Theorem 2.8** Fully dynamic maintenance of 2VCCs of an undirected graph $G$ requires $O(\log^2 n)$ time using $O(n\log(2n)/\log n)$ processors per update and $O(1)$ time with one processor per query, or requires $O(\log n \log \frac{m}{n})$ time using $O(n\log(2n)/\log n)$ processors per update and $O(\log n)$ time using $O(n\log(2n)/\log n)$ processors per query on a PRIORITY CRCW PRAM.

**Proof.** When an edge is inserted or deleted, it changes only $O(1)$ leaf nodes of $BT$. Meanwhile, the sparse 2-vertex certificate at each node in the path of $BT$ from each of these leaf nodes to the root node needs to be updated. The computation of the certificate at a node requires $O(\log n)$ time and at most $O(n\log(2n)/\log n)$ processors by Cheriyan et al's algorithm [21]. At the root node, we run Tarjan et al's algorithm [164] to maintain the 2VCCs on the graph consisting of the certificate at
this node. Their algorithm for finding all 2VCCs in a graph with \( n \) vertices and \( m \) edges requires \( \mathcal{O}(\log n) \) time and \( \mathcal{O}(m+n) \) processors which can also be implemented in \( \mathcal{O}(\log^2 n) \) with \( \mathcal{O}((m+n)/\log n) \) processors by Brent’s theorem [18]. Therefore updating all data structures requires \( \mathcal{O}(n \log \frac{m}{n} + \log^2 n) = \mathcal{O}(\log^2 n) \) time and \( \mathcal{O}(n\alpha(2n,n)/\log n) \) processors. For a query, we just check the root of \( \mathcal{B}T \) to see whether the two vertices are in the same 2VCC which costs \( \mathcal{O}(1) \) time with one processor. If we want to reduce the update time further, there is another approach in which, for each update, we just recompute the certificates for all involved nodes of \( \mathcal{B}T \), and for each query about whether there exists \( k \) vertex-disjoint paths, we run the algorithm by Klawe et al [101], at the root of \( \mathcal{B}T \). For the case where \( k = 2 \), their algorithm requires \( \mathcal{O}(\log n) \) time and \( \mathcal{O}(n\alpha(2n,n)/\log n) \) processors. So it requires \( \mathcal{O}(\log n \log \frac{m}{n}) \) time using \( \mathcal{O}(n\alpha(2n,n)/\log n) \) processors per update. \( \Box \)

2.5.3 The maintenance of 3VCCs

Assume that \( G \) is biconnected. If not, we maintain two types of data structures, one for biconnectivity of \( G \) and another for triconnectivity of \( G \). That is, if there is a query to ask whether vertices \( u \) and \( v \) are in the same 3VCC, we first check the data structures for biconnectivity to see whether \( u \) and \( v \) are in the same 2VCC. If yes, we run the procedure of query on the data structures for triconnectivity. Otherwise, the response of this query is false. By Theorem 2.8, the fully dynamic maintenance of biconnectivity can be done in \( \mathcal{O}(\log n \log \frac{m}{n}) \) time using \( \mathcal{O}(n\alpha(2n,n)/\log n) \) processors.

Theorem 2.9 Fully dynamic maintenance of 3VCCs of \( G \) requires \( \mathcal{O}(\log n \log \frac{m}{n} + \log n \log \log n/\alpha(3n,n)) \) time using \( \mathcal{O}(n\alpha(3n,n)/\log n) \) processors per update and \( \mathcal{O}(1) \) time with one processor per query. The algorithm runs in a PRIORITY CRCW PRAM.

Proof. The approach adopted is the same as used for Theorem 2.8 except (1) we find a sparse 3-vertex certificate at each involved node when there is an update; (2) we replace Tarjan et al’s [164] algorithm for biconnectivity by the algorithm for triconnectivity due to Fussel et al [61]. Their algorithm for a graph with \( n \) vertices and \( m \) edges requires \( \mathcal{O}(\log n) \) time and \( \mathcal{O}((m+n) \log \log n/\log n) \) processors, which also can be implemented in \( \mathcal{O}(\log n \log \log n/\alpha(m,n)) \) time using \( \mathcal{O}((m+n)\alpha(m,n)/\log n) \) processors by Brent’s theorem. \( \Box \)

2.6 Conclusions

In this chapter we studied the problem of fully dynamic maintenance of \( k \)-connectivity of graphs, and presented the first NC algorithm for it with \( \tilde{\mathcal{O}}(n) \) work for \( k = 2,3 \). In particular, our 2-edge connectivity algorithm uses \( o(n) \) processors only. For the biconnectivity problem, we obtained an NC algorithm with \( \mathcal{O}(n) \) processors. From this algorithm we also derived NC algorithms with \( \mathcal{O}(n) \) processors for the fully dynamic maintenance of 3-edge connectivity and triconnectivity.
Chapter 3

NC Algorithms for the Partially Dynamic Maintenance of the Solution of All Pairs Shortest Paths

3.1 Introduction

Let $G(V, E)$ be a weighted or unweighted directed (acyclic) graph. Initially, $G(V, E) = G(V, \emptyset)$ and $V = \{1, 2, \ldots, n\}$. An important problem in the algorithm area is how to maintain on-line all pairs shortest paths in $G$ after undergoing dynamic updates. The problem consists of (1) maintaining a distance matrix $Dist$, where an entry $Dist[i, j]$ in $Dist$ contains the length of the shortest path from vertex $i$ to vertex $j$; (2) finding the shortest paths and answering queries about the lengths of the paths for edge insertions, $1 \leq i, j \leq n$. In detail, in this chapter we consider how to maintain all pairs shortest paths in $G$ in parallel after undergoing an intermixed sequence of the following operations:

- $Add(x, y, w)$: insert an edge of cost $w$ from vertex $x$ to vertex $y$;
- $LengthShortestPath(x, y)$: return the length of the shortest path from $x$ to $y$ if it exists; $+\infty$ otherwise;
- $FindShortestPath(x, y)$: return the shortest path from $x$ to $y$ if it exists.

The problem of fully dynamic maintenance of all pairs shortest paths in a directed graph has been studied in the sequential context previously [9, 49, 125, 151]. Ausiello et al. [9] considered a special case of this problem in which the edge cost is bounded by an integer $C$. For this case their fully dynamic algorithm requires $O(Cn^3\log n)$ amortized time for an intermixed sequence of edge insertions and edge cost decreases, $O(1)$ time for a query $LengthShortestPath(x, y)$, and $O(l)$ time for $FindShortestPath(x, y)$ where $l$ is the number of edges of the shortest path from $x$ to $y$. Independently Lin and Chang [125] also gave the same results for this special case. For the general case, Even and Gazit [49] showed that it requires $O(n^2)$ time for an edge insertion and/or an edge cost decrease,
and $O(mn + n^2 \log n)$ time for an edge deletion and/or an edge cost increase respectively, where $m$ is the number of edges in the current graph. Rohnert [151] also considered this problem by giving the similar results. However, to the best of our knowledge, there is no known NC algorithm for the partially dynamic maintenance of all pairs shortest paths using $O(n^2)$ processors only, in spite of the existence of several efficient sequential algorithms. Without exception, all previously existing NC algorithms are based on matrix multiplication which requires $M(n)$ processors at least. Currently the best result for $M(n)$ is $n^{2.376}$ [32].

One closely related problem is the all pairs longest paths problem on a DAG which is polynomially solvable, but this problem on a general graph is NP-complete [68]. Ausiello et al. [9] extended their results on all pairs shortest paths to all pairs longest paths in a DAG, and presented an algorithm for this latter problem. Their algorithm for the problem has the same time complexity as that for the former problem.

The topological sorting and transitive closure problems have wide applications in distributed computing and compilers [175, 160]. Spacchamela et al. [160] proposed an amortized $O(mn)$ time algorithm for $m$ edge insertions. Note that there is a topological ordering of vertices in a directed graph only if the graph is a DAG. For the transitive closure problem, there are several sequential algorithms for it [89, 108, 92]. The first fully dynamic algorithm was given by Ibaraki and Katoh [89]. Their algorithm requires $O(n^3)$ time for a sequence of edge insertions, and $O(n^2(m + n))$ time for $m$ edge deletions. Later La Poutré and van Leeuwen [108] considered a special case of this problem with the restriction that the augmented graph is a DAG. For this case, their algorithm requires $O(nm)$ amortized time for either $m$ edge insertions or $m$ edge deletions but not for both of them, where $m$ is either the number of edges in the original graph for deletion or the number of edges in the resulting graph for insertion. Independently, Italiano [92] also presented an algorithm for the DAG with the same time bound. If the maximum vertex degree of the augmented graph is bounded by $\Delta$, Yellin [175] proposed an algorithm which is still the best so far. His algorithm requires $O(m\Delta)$ amortized time, where $m$ is either the number of edges of the transitive closure in the final graph for edge insertions or the number of edges in the initial graph for edge deletions.

In this chapter we suggest an NC algorithm for the partially dynamic maintenance of the solution of all pairs shortest paths problem. Our algorithm requires (i) $O(\log n)$ time and $O(n^2)$ processors for an edge insertion and/or an edge cost decrease; (ii) $O(1)$ time and $O(n^2)$ processors for the queries of the lengths up to $n^2$ pairs shortest paths; and (iii) $O(\log n)$ time and $O(n^2)$ processors for finding $n$ shortest paths. The proposed NC algorithm can also be extended to solve other dynamic update problems such as the all pairs longest paths problem on a DAG, the topological sorting problem on a DAG, and the transitive closure problem in a directed graph. Unless specified, our computational model is a CREW PRAM.
3.2 An Algorithm for the All Pairs Shortest Paths Problem

The data structure for this problem comprises two matrices Dist and PT. An entry Dist[i, j] in Dist is the length of the shortest path from vertex i to vertex j if such a path exists, +∞ otherwise. Consider the single source shortest path problem starting from i. It consists of computing the shortest paths from i to all other vertices. Denote by T(i), a shortest path tree rooted at i, where the parent FT[i](j) of vertex j is j’s immediate ancestor on the shortest path from i to j if j is reachable from i, FT[i](j) is assigned “NULL” otherwise. An entry PT[i, j] in PT is defined as PT[i, j] := FT[i](j).

Having the above data structure, we now consider inserting an edge ⟨x, y⟩ into the graph G. Let Distold[i, j] and Distnew[i, j] be the length of the shortest path from i to j before and after inserting the edge ⟨x, y⟩ respectively. Then, by the definition of shortest paths, for all i ∈ V and j ∈ V, we have

\[
\text{Distnew}[i, j] := \min \{\text{Distold}[i, j], \, \text{Distold}[i, x] + \text{Distold}[y, j] + w(x, y)\} \tag{3.1}
\]

where w(x, y) represents the cost of the edge ⟨x, y⟩.

By Equation 3.1, a simple dynamic update algorithm for the all pairs shortest paths problem is presented as follows. For each T(i), we first assigned a copy Ti,y of tree T(y) to it. Then for each tree T(i) and every vertex j ∈ V, we check whether Distold[i, j] > Distold[i, x] + Distold[y, j] + w(x, y). If yes, we assign Dist[i, j] by Distold[i, x] + Distold[y, j] + w(x, y), and delete the edge (j, FT[i](j)) from T(i), i.e., set FT[i](j) := j. Otherwise we delete the edge (j, FT[i](j)) from the tree Ti,y. Let T′(i) and T′i,y be the resulting graphs after deleting the edges from T(i) and Ti,y respectively. We now compute the connected components of graphs T′(i) and T′i,y. Denote by CC_T(q) the connected component containing vertex q in G. Let F_{T}^{old}(i) and F_{T}^{new}(i) be the parent of vertex i in a tree T before and after inserting ⟨x, y⟩ respectively. Now we check whether Distold[i, y] > Distnew[i, y]. If yes, x is the parent of y in T(i) after inserting ⟨x, y⟩, and we set F_{T}^{new}(y) := x. Finally we merge two connected components CC_{T}(i) and CC_{T′}(y) (here actually two subtrees) into a connected component, i.e., we update the tree T(i). This update is proceeded as follows. Initially, we set F_{T}^{new}(v) := NULL for all v ∈ V. Then, for each vertex k ∈ CC_{T}(i), we set F_{T}^{new}(k) := F_{T}^{old}(k), and for each k ∈ CC_{T′}(y), we set F_{T}^{new}(k) := F_{T,i,y}^{old}(k).

The algorithm above can be easily modified to process the case of edge cost decrease. We omit it here. In the following we show that the proposed algorithm is correct.

**Lemma 3.1** For any vertex k ≠ y in T(y), k ∈ CC_{T,i,y}(y) if and only if

\[
\text{Distold}[i, k] > \text{Distold}[i, x] + \text{Distold}[y, k] + w(x, y). \tag{3.2}
\]

**Proof.** For any k ∈ V which is reachable from y, if k does not satisfy Inequality 3.2, the edge (k, FT[i](k)) will be deleted from T(i), and hence k is excluded from CC_{T,i,y}. So, every remaining vertex in CC_{T,i,y} must satisfy Inequality 3.2.
Let $k$ be such a vertex that $\text{Dist}_{\text{old}}[i,k] > \text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,k] + w(x,y) = \text{Dist}_{\text{new}}[i,k]$. In order to prove $k \in CC_{T_i,y}(y)$, we just need to show that, for every ancestor $p$ of $k$ in $T(y)$,

$$\text{Dist}_{\text{old}}[i,p] > \text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,p] + w(x,y).$$

(3.3)

We prove this by contradiction. Assume that vertex $p_1$ is the first ancestor vertex (starting from $k$) of $k$ which does not satisfy Inequality 3.3. Then, $\text{Dist}_{\text{old}}[i,p_1] \leq \text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,p_1] + w(x,y)$. However, we already knew that $\text{Dist}_{\text{old}}[i,k] \leq \text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,p_1,k] \leq (\text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,p_1] + w(x,y)) + \text{Dist}_{\text{old}}[p_1,k] = \text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,k] + w(x,y) = \text{Dist}_{\text{new}}[i,k]$. Contradiction. So, for every ancestor $p$ of $k$ in $T(y)$, $\text{Dist}_{\text{old}}[i,p] > \text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,p] + w(x,y)$. From this we derive that the vertices $k$ and $y$ are in the same connected component of $T_i^t$, i.e., $k \in CC_{T_i,y}(y)$. □

Using the same argument we have the following lemma.

**Lemma 3.2** For any vertex $k \neq i$ in $T(i)$, $k \in CC_{T_i}^t(i)$ if and only if

$$\text{Dist}_{\text{old}}[i,k] \leq \text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,k] + w(x,y).$$

(3.4)

**Lemma 3.3** The proposed algorithm is correct.

**Proof.** Let $k$ be a vertex in $CC_{T_i}^t(i)$. By Lemma 3.2, $\text{Dist}_{\text{old}}[i,k] \leq \text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,k] + w(x,y)$. So, the edge $(x,y)$ added to $G$ does not affect the shortest path from $i$ to $k$. If $y$ in $T(i)$ already changed its parent to $x$, and $k$ in $CC_{T_i}^t(y)$, by Lemma 3.1 $\text{Dist}_{\text{old}}[i,k] > \text{Dist}_{\text{old}}[i,x] + \text{Dist}_{\text{old}}[y,k] + w(x,y)$. This means that the shortest path from $i$ to $k$ must pass through the edge $(x,y)$. The parent of $k$ in $T(y)$ therefore is also the parent of $k$ in $T(i)$ after inserting $(x,y)$. □

**Lemma 3.4** The update algorithm for inserting an edge $(x,y)$ into $G$ requires $O(\log n)$ time and $O(n^2)$ processors.

**Proof.** Note that the computation of connected components in a forest with $n$ vertices can be done in $O(\log n)$ time using $O(n)$ processors. Therefore, the entire update for an edge insertion requires $O(\log n)$ time and $O(n^2)$ processors because there are $n$ shortest path trees. For each such a tree $T(i)$, it may need to merge with a subtree of $T(y)$ including $y$. The merge needs $O(\log n)$ time and $O(n)$ processors, $1 \leq i \leq n$. So, the algorithm for maintaining all pairs shortest paths requires $O(\log n)$ time and $O(n^2)$ processors in total. □

Now consider the operation $\text{LengthShortestPath}(u,v)$. It can be handled easily by retrieving the value of entry $\text{Dist}(u,v)$. So, we are able to answer as many as $n^2$ such queries in $O(1)$ time because there are $n^2$ processors available. The query for finding the shortest path from $u$ to $v$ can be done by finding the directed path from $v$ to the
root \( u \) in the tree \( T(u) \), which can be obtained easily in \( O(\log n) \) time using \( O(n) \) processors by path doubling technique in parallel processing. Therefore we have the following theorem.

**Theorem 3.1** Given a weighted directed graph \( G(V,E) \) provided that edge insertions introduce no negative cycles in \( G \), there exists a data structure which supports \( n^2 \) \( \text{LengthShortestPath}(x,y) \) operations in \( O(1) \) time, \( n \) \( \text{FindShortestPath}(x,y) \) operations in \( O(\log n) \) time, and an \( \text{Add}(x,y,w) \) operation in \( O(\log n) \) time, using \( O(n^2) \) processors at most.

### 3.3 Algorithms for Other Problems

#### 3.3.1 Finding all pairs longest paths in a DAG

Following the similar discussion by Ausiello et al. [9], we can modify the algorithm in Section 3.2 to make it capable of performing the following operations on a DAG. Assume that the augmented graph is still a DAG.

- \( \text{Add}(x,y,w) \): insert an edge of cost \( w \) from \( x \) to \( y \);
- \( \text{LengthLongestPath}(x,y) \): return the length of the longest path from \( x \) to \( y \) if it exists; \( +\infty \) otherwise;
- \( \text{FindLongestPath}(x,y) \): return the longest path from \( x \) to \( y \) if it exists.

The data structure for this problem is a modification of that for the all pairs shortest paths problem. That is, for each \( i \in V \) we establish a directed tree \( T(i) \) rooted at vertex \( i \). Define the parent \( F_T(i)(j) \) of a vertex \( j \) in \( T(i) \) by \( u \), where \( u \) is the immediate ancestor of \( j \) on the longest path from \( i \) to \( j \). The tree \( T(i) \) is stored in the \( i \)th row of the matrix \( PT \) defined in Section 3.2. Furthermore a matrix \( Long \) is also established, in which an entry \( Long[i,j] \) is the length of the longest path from \( i \) to \( j \) if it does exist, otherwise \( Long[i,j] \) is \(-\infty\).

When inserting an edge \( \langle x,y \rangle \), we first check whether it introduces a cycle. This can be done easily by checking whether \( Long[y,x] > 0 \). If yes, the edge \( \langle x,y \rangle \) cannot be added to \( G \), because a directed cycle in \( G \) will be formed consisting of the longest path from \( y \) to \( x \) and the edge \( \langle x,y \rangle \) itself. Otherwise, we re-compute some longest paths after inserting \( \langle x,y \rangle \). Assume that there is no cycle introduced by adding \( \langle x,y \rangle \) to the current graph. Let \( Long_{old}[i,j] \) and \( Long_{new}[i,j] \) be the length of the longest path from \( i \) to \( j \) before and after inserting \( \langle x,y \rangle \). Then, by the definition of the longest paths, for all \( i \in V \) and \( j \in V \), we have

\[
Long_{old}[i,i] := Long_{new}[i,i] := 0; \quad (3.5)
\]

\[
Long_{new}[i,j] := \max\{Long_{old}[i,j], Long_{old}[i,x] + Long_{old}[y,j] + w(x,y)\}. \quad (3.6)
\]

The processing for this problem is similar to that used for the all pairs shortest paths problem. Therefore, we have
Theorem 3.2 Given a DAG $G(V, E)$ provided edge insertions introduce no cycles, there exists a data structure which supports $n^2$ $\text{LengthLongestPath}(x, y)$ operations in $O(1)$ time, $n$ $\text{FindLongestPath}(x, y)$ operations in $O(\log n)$ time, and an $\text{Add}(x, y, w)$ operation in $O(\log n)$ time. The number of processors used is $O(n^2)$.

3.3.2 Topological sorting in a DAG

A topological order of a partially ordered set $(V, \Omega)$ is a total order $ord$ on $V$ such that if $(x, y) \in \Omega$, then $ord(x) < ord(y)$. It is straightforward to represent a partially ordered set $(V, \Omega)$ by a DAG $G(V, E)$ such that $(x, y) \in E$ if and only if $(x, y) \in \Omega$. A topological order $ord$ of vertices in $G$ is to map each vertex $v$ to an integer $ord(v)$ between 1 and $n$ such that, for each $(x, y) \in E$, $ord(x) < ord(y)$. The integer $ord(u)$ is called the topological numbering of $u$. A simple sequential algorithm for this problem requires $O(n + m)$ time by using depth-first search technique [132]. But all known parallel algorithms for this problem [19, 163] are based on matrix multiplication which requires $O(n \log^2 n)$ time and $M(n)$ processors at least on a CREW PRAM.

In the following, we deal with maintaining the topological numbering of vertices dynamically after a sequence of edge insertion operations. The data structures for this problem include two matrices $\text{Dist}$ and $\text{PT}$, and two one-dimensional arrays $ord$ and $\text{Vex}$, where an entry $\text{Dist}[i][j]$ in $\text{Dist}$ is the length of the shortest path from $i$ to $j$ if $j$ is reachable from $i$, $\text{Dist}[i][j] := +\infty$ otherwise. An entry $ord(i)$ in $ord$ represents the topological numbering of vertex $i$, and an entry $\text{Vex}(i)$ in $\text{Vex}$ represents the vertex whose topological numbering is $i$. For each vertex $i \in V$ we build a reachability tree $T(i)$ which actually is a shortest path tree starting at $i$, and $T(i)$ is stored in the $i$th row of the matrix $\text{PT}$.

Consider inserting an edge $(x, y)$ to a DAG $G$. We first check whether the augmented graph is still a DAG, i.e., whether this operation introduces any cycle in $G$. This can be done by checking if $\text{Dist}[y][x] \neq +\infty$. If $\text{Dist}[y][x] \neq +\infty$, we cannot add this edge to $G$, because otherwise a directed cycle will be introduced. From now on, assume that inserting an edge $(x, y)$ introduces no cycles. We then proceed as follows. Check whether the old topological order is still valid, i.e., whether $ord(x) < ord(y)$ is still holds. If so, we just update the matrices $\text{Dist}$ and $\text{PT}$ by the algorithm in Section 3.2, and do nothing about the topological order update. Otherwise, we must update $ord$ and $\text{Vex}$ too, i.e., recompute the topological numbering of vertices. Our algorithm for this latter case stems from the following lemma, proven by Spaccamela et al.

Lemma 3.5 [160] Given a DAG $G(V, E)$ and a topological order $ord$ of its vertices, let $G'(V, E \cup \{(x, y)\})$ be the graph obtained from $G$ by inserting edge $(x, y)$. If $G'$ is acyclic, then there is a topological order $ord_{new}$ of $G'$ such that $ord_{new}(w) = ord_{old}(w)$ for all $w$ with $ord_{old}(w) < ord_{old}(x)$ or $ord_{old}(w) > ord_{old}(y)$, where $ord_{old}(w)$ is the topological numbering of vertex $w$ before inserting $(x, y)$.

Let $X = \{x_1, x_2, \ldots, x_p\}$ be a vertex set such that $x$ is reachable from each vertex in $X$, and $ord(y) \leq ord(x_i) \leq ord(x)$ for all $i$, $1 \leq i \leq p$, where $ord(x_i) < ord(x_{i+1})$ for
all $i \leq p - 1$. The construction of $X$ can be done in $O(\log n)$ time using $O(n)$ processors by checking whether $\text{Dist}(u, x) \neq +\infty$ and $\text{ord}(y) \leq \text{ord}(u) \leq \text{ord}(x)$ for each $u$, and then by sorting the selected vertices.

Let $Y = \{y_1, y_2, \ldots, y_q\}$ be a vertex set in which each vertex is reachable from $y$ and $\text{ord}(y) \leq \text{ord}(y_j) \leq \text{ord}(x)$ for all $q, 1 \leq j \leq q$, where $\text{ord}(y_j) < \text{ord}(y_{j+1})$ for all $j \leq q - 1$. Obviously, $Y$ can be obtained by checking each vertex $u$ in $T(y)$ if it satisfies $\text{ord}(y) \leq \text{ord}(u) \leq \text{ord}(x)$.

It is easy to show that $X \cap Y = \emptyset$ when $X \neq \{x\}$ and $Y \neq \{y\}$, since otherwise the augmented graph is no longer a DAG. Let $A$ and $B$ be two disjoint vertex sets. Define $\text{ord}(A) < \text{ord}(B)$ if $\text{ord}(u) < \text{ord}(v)$ for every $u \in A$ and every $v \in B$.

By Lemma 3.5, the topological numbering $\text{ord}(u)$ of vertex $u$ may need to be updated if $\text{ord}(y) \leq \text{ord}(u) \leq \text{ord}(x)$. Let $U = \{u \mid \text{ord}(y) \leq \text{ord}(u) \leq \text{ord}(x)\}$. Obviously, $X \subseteq U$ and $Y \subseteq U$. Let $W = U - Y$. Then, $x \in W$ and $y \in Y$. In order to compute the new topological order after inserting $(x, y)$, each of the following conditions must be satisfied: (i) $\text{ord}_{\text{new}}(x) < \text{ord}_{\text{new}}(y)$ and $\text{ord}_{\text{new}}(W) < \text{ord}_{\text{new}}(Y)$ because $\text{ord}_{\text{old}}(x) = \max\{\text{ord}_{\text{old}}(v) \mid v \in W\}$ and $\text{ord}_{\text{old}}(y) = \min\{\text{ord}_{\text{old}}(u) \mid u \in Y\}$; and $\text{ord}_{\text{old}}(x) > \text{ord}_{\text{old}}(y)$; (ii) for any two vertices $u, v \in W$, $\text{ord}_{\text{new}}(u) < \text{ord}_{\text{new}}(v)$ if $\text{ord}_{\text{old}}(u) < \text{ord}_{\text{old}}(v)$; and (iii) for any two vertices $u, v \in Y$, $\text{ord}_{\text{new}}(u) < \text{ord}_{\text{new}}(v)$ if $\text{ord}_{\text{old}}(u) < \text{ord}_{\text{old}}(v)$.

Based on the discussion above, the topological numbering of vertices in $U$ is updated as follows. First, mark all vertices in $Y$ which can be implemented by an $1$-dimensional array $\text{count}$, where the entry $\text{count}(i) = 1$ if $\text{Vex}(i)$ is marked, otherwise $\text{count}(i) = 0$. Then, compute the partial prefix sum $\text{sum}(i) := \sum_{j=1}^{i} \text{count}(j)$ of array $\text{count}$ for all $i, 1 \leq i \leq n$. Finally, assign new topological numberings to all vertices. For each $u \in U$, $\text{ord}_{\text{new}}(u) := \text{ord}_{\text{old}}(u)$ and $\text{Vex}_{\text{new}}(\text{ord}_{\text{old}}(u)) := u$. Let $t = \text{ord}_{\text{old}}(y)$, $h = \text{ord}_{\text{old}}(x)$, and $k = \max\{\text{sum}(i) \mid 1 \leq i \leq n\}$. For each $i$ between $l$ and $h$, if $\text{Vex}(i) \in Y$, then $\text{Vex}_{\text{new}}(h-k+\text{sum}(i)) := \text{Vex}_{\text{old}}(i)$ and $\text{ord}_{\text{new}}(\text{Vex}_{\text{old}}(i)) := h-k+\text{sum}(i)$; if $\text{Vex}(i) \in W$, then $\text{Vex}_{\text{new}}(i-\text{sum}(i)) := \text{Vex}_{\text{old}}(i)$ and $\text{ord}_{\text{new}}(\text{Vex}_{\text{old}}(i)) := i-\text{sum}(i)$.

It is not difficult to verify that the new topological numbering of vertices satisfies Conditions (i)–(iii). The computation of all prefixes of $\text{count}$ can be done in $O(\log n)$ time using $O(n/\log n)$ processors. Thus, we have the following theorem.

**Theorem 3.3** Given a DAG $G(V, E)$ provided edge insertions introduce no cycles, there exist a data structure which supports maintaining the topological order of vertices in $G$ when there is an edge insertion. The proposed algorithm requires $O(\log n)$ time and $O(n^2)$ processors.

### 3.3.3 The dynamic transitive closure problem

The problem is to compute the transitive closure of an augmented graph dynamically undergoing an intermixed sequence of the following operations:

- **Add**(x, y): insert an edge from x to y;
- **Reachability**(x, y): return TRUE if y is reachable from x, otherwise FALSE;
• ReachabilityPath(x,y): return a directed path from x to y if such one exists.

By the similar discussion as in the previous section, we maintain two matrices Reach and PT, where an entry Reach[i,j] in Reach is TRUE if vertex j is reachable from vertex i, and FALSE otherwise. The ith row of PT stores a directed rooted tree T(i), where T(i) is the shortest path tree from i to all other vertices which are reachable from i. Note that the edge cost of each edge is one. So, the transitive closure problem is a special case of the all pairs shortest paths problem. We therefore have the following theorem.

**Theorem 3.4** Given a directed graph G(V,E), there exists a data structure which supports n^2 Reachability(x,y) operations in O(1) time, n ReachabilityPath(x,y) operations in O(log n) time, and an Add(x,y) operation in O(log n) time. The number of processors used is O(n^2).

**Remark.** The above parallel algorithms require the same time for processing a single query and multiple queries. It should be noticed that it is more efficient for parallel dynamic algorithms to process a batch of queries rather than each query individually. The possibility for processing multiple queries simultaneously is provided by the fact that queries only retrieve information from the given data structures without modifying the underlying data structures.

### 3.4 Conclusions

In this chapter we discussed the partially dynamic update problem related to directed (acyclic) graphs such as the all pairs shortest paths problem, the all pairs longest paths problem, the topological sorting problem, and the transitive closure problem. Our algorithm requires O(log n) time for an edge insertion, O(1) time for n^2 queries of a simple property of graphs such as the length of the shortest (longest) paths (in a DAG) and the reachability between a pair of vertices, and O(log n) time for both finding n pairs shortest (longest) paths and maintaining the topological order of vertices. The number of processors used is at most O(n^2).
NC Algorithms for Testing $k$-Connectivity of Directed and Undirected Graphs with Fixed $k$

4.1 Introduction

Connectivity of graphs (vertex connectivity as well as edge connectivity) is considered to be one of the classic subjects in graph theory, and has many practical applications, e.g., in reliability of communication networks, chip and circuit design, and cluster analysis. Designing efficient parallel algorithms for testing graph connectivity is clearly a basic problem in parallel computation. In this chapter we present parallel algorithms for vertex and edge connectivity for each of directed and undirected graphs. In all four cases our algorithms have better theoretical complexity than previously published algorithms. The model of parallel computation used is a PRIORITY CRCW PRAM.

We will write our complexity measures in terms of three basic quantities. $P(n, m)$ is the number of processors needed to find a directed path in time $O(\log n)$ between two specified vertices in a directed graph with $O(n)$ vertices and $O(m)$ edges. Secondly, $T(n, m)$ is the number of processors needed to determine the set of vertices reachable from a specified vertex in a directed graph with $O(n)$ vertices and $O(m)$ edges, in time $O(\log n)$. The current best results for $P(n, m)$ and $T(n, m)$ are both $O(n^{2.376})$, using matrix multiplication [32]. Finally, $C(n, m)$ is the number of processors needed to find the connected components of an undirected graph with $n$ vertices and $m$ edges in time $O(\log n)$. In this case the current best result is $C(n, m) = (n + m)\alpha(m, n)/\log n$ [31].

First consider the case of directed graphs. To our knowledge, there are no published NC algorithms for testing $k$-vertex connectivity or $k$-edge connectivity for any $k$ except $k = 1$. We give algorithms taking time $O(k \log n)$, using $nP(n, m)$ processors in the case of edge connectivity, and $(n + k^2)P(n, m)$ processors in the case of vertex connectivity. These algorithms use simple implementations of the Ford-Fulkerson network flow algorithm. In the case where the connectivity is less than $k$, we also find separating sets within the same time bound. The best deterministic sequential algorithm for edge-connectivity is due to Gabow [62] and takes time $O(\lambda m \log(n^2/m))$ using a matroid approach where $\lambda$ is the value of the edge connectivity.
Next consider the case of undirected graphs. Khuller and Schieber [101] present algorithms for testing \( k \)-edge connectivity and \( k \)-vertex connectivity using time \( O(k^2 \log n) \). The required number of processors is \( O(nkC(n,m)) \) and \( O((nk+k^3)C(n,m)) \), respectively. The number of processors needed in the latter case was recently reduced to \( O((nk+k^3)C(n,m)) \) by Cheriyan et al. [21], using the sparse \( k \)-vertex certificate technique. Our algorithms improve the time requirement to \( O(k \log n) \) at some expense to the number of processors. Clearly, if an undirected edge is considered to be a pair of oppositely directed edges, our algorithms for directed graphs can be used with the same complexity. By using sparse \( k \)-edge and \( k \)-vertex certificates we can reduce the number of processors to \( O(nP(n,nk)) \) for \( k \)-edge connectivity and \( O((n+k^2)P(n,nk)) \) for \( k \)-vertex connectivity. The best deterministic sequential algorithm for testing \( k \)-vertex connectivity takes time \( O(nm) \) for fixed \( k \) [47]. For \( k \)-edge connectivity, the above mentioned algorithm of Gabow can be used, but for some values of the parameters one of the two algorithms of Matula [131] is better, as they use time \( O(nm) \) and \( O(\lambda n^2) \) respectively.

We also show that the \( k \)-edge connectivity problem is NC-reducible to the \( k \)-vertex connectivity problem for both undirected graphs and directed graphs, with \( k \) arbitrary. Therefore, if the solution for the \( k \)-vertex connectivity problem is in NC, then the \( k \)-edge connectivity problem is also in NC. It must be mentioned that Galli et al. [65] also gave such a reduction on undirected graphs before.

The chapter is organized as follows. In Section 4.2 we give our algorithms for directed graphs, and in Section 4.3 we give our improvements for undirected graphs. In Section 4.4 we discuss the NC-reduction of \( k \)-edge connectivity to \( k \)-vertex connectivity. A conclusion is followed in Section 4.5.

### 4.2 Directed graphs

Let \( G(V,E) \) be a directed graph. Multiple edges will not be allowed, but could be incorporated without much effort. Let \( s \) and \( t \) be distinct vertices in \( G \) that are not connected by an edge. An \( s \rightleftharpoons t \) **vertex separator** is a subset \( S \subseteq V \setminus \{s,t\} \), such that every path from \( s \) to \( t \) contains at least one vertex from \( S \). Define \( N(s,t) \) to be the minimum cardinality of an \( s \rightleftharpoons t \) vertex separator. By Menger’s Theorem, \( N(s,t) \) is also the maximum number of vertex disjoint paths from \( s \) to \( t \). The **vertex connectivity** \( \kappa(G) \) of \( G \) is defined to be \( \kappa(G) = \min \{N(s,t) \mid s,t \in V, s \neq t, (s,t) \notin E\} \).

Similarly, let \( s \) and \( t \) be distinct vertices of \( G \) and define an \( s \rightleftharpoons t \) **edge separator** to be a set \( Q \subseteq E \) such that every path from \( s \) to \( t \) uses at least one edge in \( Q \). Define \( M(s,t) \) to be the minimum cardinality of an \( s \rightleftharpoons t \) edge separator. By Menger’s Theorem, \( M(s,t) \) is also the maximum number of edge disjoint paths from \( s \) to \( t \). The **edge connectivity** of \( G \) is defined to be \( \lambda(G) = \min \{M(s,t) \mid s,t \in V, s \neq t\} \).

Our fundamental approach will be the use of network flows to find disjoint paths. Consider the directed graph \( G(V,E) \) to be a 0–1 network in which the capacity of each edge is one. Suppose \( G \) carries some 0–1 legal \( s \rightleftharpoons t \) flow \( f \). Define the auxiliary directed graph \( \tilde{G} = G(V,E) \) as follows: For each distinct \( u,v \in V \), \( \langle u,v \rangle \in E \) if and only if either \( \langle u,v \rangle \in E \) and \( f(u,v) = 0 \), or \( \langle v,u \rangle \in E \) and \( f(v,u) = 1 \). Note that \( \tilde{G} \) is
Procedure \texttt{Flow}(G, k, i, s, t);
/* \(P\) is the set of edges with flow 1; and \(i\) is actual flow value */
Initialize \(P := \emptyset\);
for \(i := 0\) to \(k\) do
  Form \(\hat{G} = G(V, \hat{E})\), where \(\hat{E} := (E - P) \cup \{(u, v) \mid (v, u) \in P\}\);
  if there is no directed \(s-t\) path \(P'\) in \(\hat{G}\) then \textbf{return} FAIL;
  else \(P := P \cup P' - \{(u, v), (v, u) \mid (u, v) \in P\ \text{and} (v, u) \in P'\}\)
endfor
return \(P\).

similar to the "residue network" of \(G\) and \(f\), but has only edges of capacity one. It is still true that paths in \(\hat{G}\) are augmenting paths in \(G\).

**Theorem 4.1** The flow \(f\) is a maximum \(s-t\) flow in \(G\) if and only if \(\hat{G}\) has no directed \(s-t\) paths.

**Proof.** See Chapter 6 in [48].

This theorem implies an algorithm for finding an \(s-t\) flow of total value \(k\), or proving there is none. The details are as follows.

**Lemma 4.1** Given a 0-1 network \(G(V, E)\) and \(s, t \in V\), testing whether the value of the flow from \(s\) to \(t\) is no less than \(k\) can be done in \(O(k \log n)\) time with \(P(n, m)\) processors.

**Proof.** The construction of \(\hat{G}\) can be done in \(O(\log n)\) time using \((m + n)/\log n\) processors. Finding an \(s-t\) path requires \(O(\log n)\) time and \(P(n, m)\) processors. The updating of \(P\) can be finished in \(O(\log n)\) time using \(O(kn)\) processors, as follows: firstly we sort the edges in \(P\) by their key \((u, v)\), which costs \(O(\log n)\) time and \(O(kn)\) processors because \(|P| \leq kn\). Then for each edge \((u, v) \in P'\), we look up the sorted list on \(P\) by binary searching to see whether \((v, u)\) is in \(P\). If it does, delete edges \((u, v)\) and \((v, u)\) from these two lists, respectively. This step requires \(O(\log n)\) time and \(O(n)\) processors because \(|P'| \leq n\). The remaining elements in these two lists are merged into a new list \(P\).

**Lemma 4.2** If algorithm \texttt{Flow} fails because the maximum flow is \(k' < k\), an edge separator of size \(k'\) can be found using an additional amount of \(O(\log n)\) time and \(\max\{m, T(n, m)\}\) processors.

**Proof.** Consider the final value of \(\hat{G}\) constructed by the algorithm. In time \(O(\log n)\) using \(T(n, m)\) processors, compute the set \(W\) of vertices reachable from \(s\) in \(\hat{G}\). Then, by standard flow theory, \(Q = \{(u, v) \in E \mid u \in W, v \notin W\}\) is an edge separator of size \(k'\).
**Theorem 4.2** There is an algorithm to test whether a directed graph is $k$-edge connected, and if not to find an edge separator of size less than $k$. The running time is $O(k \log n)$ and the number of processors is $O(nP(n,m))$.

**Proof.** Let $V = \{v_1, v_2, \ldots, v_n\}$. By a theorem of Schnorr [155], the edge connectivity of $G$ is $\lambda(G) = \min \{ M(v_i, v_{i+1}) \mid 1 \leq i \leq n \}$, where $v_{n+1} = v_1$. We can use procedure Flow to test in parallel if the $n$ associated flow problems all have solution at least $k$. If one does not, we can find an edge separator as described in Lemma 4.2. Obviously $\max \{ m, T(n,m) \} = O(nP(n,m))$, so $O(nP(n,m))$ processors will suffice. $\square$

We now consider $k$-vertex connectivity for directed graphs. The following lemma was inspired by a similar lemma of Even [47] for undirected graphs.

**Lemma 4.3** Suppose $V = \{v_1, v_2, \ldots, v_n\}$. Define two auxiliary directed graphs $G' = G(V', E')$ and $G'' = G(V'', E'')$ as follows. $V' = V'' = V \cup \{ z \}$; $E' = E \cup \{ (z, v_i) \mid 1 \leq i \leq k \}$; $E'' = E \cup \{ (v_i, z) \mid 1 \leq i \leq k \}$. Then $\kappa(G) \geq k$ if and only if $N(v_i, v_j) \geq k$ in $G$ for $1 \leq i \neq j \leq k$, $N(z, v_j) \geq k$ in $G'$ for $k + 1 \leq j \leq n$, and $N(v_j, z) \geq k$ in $G''$ for $k + 1 \leq j \leq n$.

**Proof.** If $\kappa(G) < k$, then we can partition $V$ into non-empty subsets $V = S \cup W \cup T$ such that $|W| < k$ and $W$ is an $s$-$t$ vertex separator for any $s \in S$ and $t \in T$. If $V_k = \{v_1, v_2, \ldots, v_k\}$ intersects both $S$ and $T$, we have $N(s,t) < k$ in $G$ for any $s \in S \cap V_k$, $t \in T \cap V_k$. If $V_k \subseteq S \cup W$, $N(z,t) < k$ in $G'$ for any $t \in T - V_k$. If $V_k \subseteq W \cup T$, $N(s,z) < k$ in $G''$ for any $s \in S - V_k$. $\square$

In order to apply Lemma 4.3 we use the standard method for finding vertex-disjoint paths with the help of flows. Define the auxiliary directed graph $\overline{G} = G(V, \overline{E})$, where $\overline{V} = \{v', v'' \mid v \in V\}$, and $\overline{E} = \{ (v', v'') \mid v \in V \} \cup \{ (u', u''), (v'', u') \mid (u,v) \in E\}$.

**Lemma 4.4** For any distinct vertices $s,t \in G$, the maximum number of vertex-disjoint $s$-$t$ paths in $G$ equals the maximum number of edge-disjoint $s'$-$t'$ paths in $\overline{G}$.

**Proof.** See Chapters 5 and 6 in [48]. $\square$

**Theorem 4.3** There is an algorithm to test whether a directed graph is $k$-vertex connected, and if not to find a vertex separator of size less than $k$. The running time is $O(k \log n)$ and the number of processors is $O((n + k^2)P(n,m))$.

**Proof.** The $k(k - 1) + 2(n - k)$ disjoint path problems defined in Lemma 4.3 can be solved in parallel using Lemma 4.4. We can form the three networks in $O(1)$ time using $m + n$ processors and solve the flow problems in $O(k \log n)$ time using $P(n,m)$ processors each, with procedure Flow. Since obviously $m + n = O(nP(n,m))$, the total number of processors required is $O((n + k^2)P(n,m))$. 
If all the flows have value at least \( k \), we know that \( G \) is \( k \)-vertex connected. Otherwise it is easy to find a small vertex separator. Suppose \( G^* = G(V^*, E^*) \) is the network with \( N(s', t') < k \). Let \( W^* \) be the set of vertices reachable from \( s'' \) in \( G(V^*, E^*_1 \cup E) \), where \( E^*_1 \) is the subset of \( E^* \) consisting of edges of the form \( \langle u'', v' \rangle \), and \( G(V^*, E) \) is the final graph \( \tilde{G} \) made by procedure \( \text{Flow} \). Then the set of all vertices \( v \in V \) such that \( v' \in W^* \) and \( v'' \notin W^* \) is an \( s-t \) vertex separator in \( \tilde{G} \) with size less than \( k \). Finding it with the same number of processors in \( O(\log n) \) time is easy. \( \square \)

### 4.3 Undirected graphs

An undirected graph can be considered as a directed graph in which the edges come in pairs \( \langle u, v \rangle, \langle v, u \rangle \). Under that interpretation it is easy to see that the definitions of connectivity and separators given in the previous section correspond to the normal definitions given for undirected graphs. Consequently, Theorems 4.2 and 4.3 apply equally to undirected graphs. However, we can reduce the required number of processors by using "the sparse \( k \)-edge and \( k \)-vertex certificates" in Section 2.2 of Chapter 2.

Define \( E_0 = \emptyset \), and let \( E_i \) be the edge set of a spanning forest by applying scan-first search on the graph \( G(V, E - E_1 - \cdots - E_{i-1}) \) for \( i = 1, \ldots, k \). Then define \( G_k = G(V, E_1 \cup E_2 \cup \cdots \cup E_k) \).

**Lemma 4.5** \( G_k \) is a sparse certificate for both \( k \)-vertex connectivity and \( k \)-edge connectivity, and \( G_k \) can be found in \( O(k \log n) \) time using \( C(n, m) \) processors.

**Proof.** For \( k \)-edge connectivity, this was proved in [135]; in fact we can use any maximal spanning forests, not necessarily the spanning forest generated by the scan-first search. For \( k \)-vertex connectivity, \( G_k \) is a sparse \( k \)-vertex certificate of \( G \) which was proved, and can be constructed in \( O(k \log n) \) time using \( C(n, m) \) processors in [21]. \( \square \)

**Theorem 4.4** There is an algorithm to test whether an undirected graph is \( k \)-edge connected, and if not to find an edge separator of size less than \( k \). The running time is \( O(k \log n) \) and the number of processors is \( O(nP(n, nk)) \).

**Proof.** Find a sparse \( k \)-edge certificate of \( G \), and test it as in Theorem 4.2. The required number of processors is \( \max\{C(n, m), nP(n, nk)\} \). However, as indicated in the introduction, \( C(n, m) = O((n + m)\alpha(m, n)/\log n) \) and also clearly \( P(n, nk) = \Omega(nk/\log n) \). Hence the term \( nP(n, nk) \) dominates. \( \square \)

**Theorem 4.5** There is an algorithm to test whether an undirected graph is \( k \)-vertex connected, and if not to find a vertex separator of size less than \( k \). The running time is \( O(k \log n) \) and the number of processors is \( O((n + k^2)P(n, nk)) \).

**Proof.** Use the same approach as for Theorem 4.4. \( \square \)
4.4 NC reduction of edge connectivity to vertex connectivity

Given the results in the previous sections, we know that $k$-edge connectivity and $k$-vertex connectivity are in NC for $k = O(\log^c n)$, where $c$ is any constant. The situation for arbitrary $k = k(n)$ remains unsolved. In this section we note that, in order to prove that $k$-edge connectivity is in NC, it would suffice to prove that $k$-vertex connectivity is in NC, both for directed and for undirected graphs.

Theorem 4.6 Let $G$ be an undirected graph, and let $L(G)$ be the line-graph of $G$. Then $\lambda(G) = \min\{\delta(G), \kappa(L(G))\}$ where $\delta(G)$ is the minimum degree of $G$.

Proof. By the definition of $L(G)$, a $k$-vertex separator of $L(G)$ is a $k$-edge separator of $G$. Conversely, a minimum $k$-edge separator of $G$ is either a $k$-vertex separator of $L(G)$ or corresponds to the edges incident with a single vertex of $G$. □

Essentially the same method works for directed graphs, if we are careful to use the correct line-graph. For a directed graph $G(V, E)$ define $L(G)$ to be the directed graph with vertex set $E$, with an edge from $\langle u, v \rangle$ to $\langle u', v' \rangle$ exactly when $v = u'$.

Theorem 4.7 Let $G$ be a directed graph, and let $L(G)$ be the line-graph of $G$. Then $\lambda(G) = \min\{\delta^+(G), \delta^-(G), \kappa(L(G))\}$ where $\delta^+(G)$ and $\delta^-(G)$ are the minimum in-degree and minimum out-degree of $G$, respectively.

Proof. A $k$-vertex separator of $L(G)$ is a $k$-edge separator of $G$. Conversely, a minimum $k$-edge separator of $G$ is either a $k$-vertex separator of $L(G)$ or corresponds to all the edges entering a single vertex of $G$ or all the edges leaving a single vertex of $G$. □

In both the undirected and directed cases, the construction of $L(G)$ can be achieved in time $O(\log n)$ using $O(n + m^2)$ processors.

4.5 Conclusion

In this chapter we presented NC algorithms for testing $k$-connectivity of directed graphs as well undirected graphs provided that $k$ is constant. But, for general $k$, whether this problem is in NC is still yet unknown.
NC Approximation Algorithms for the Optimal $k$-Connectivity Augmentation Problem with Fixed $k$

5.1 Introduction

The robustness of a communications network subject to link failure is often modeled by the edge connectivity of the associated graph. Consequently there is interest in knowing how to increase the connectivity by adding further edges, with some cost measure minimized. This “connectivity augmentation problem” has received a good deal of attention during the past two decades, [46, 152, 56, 84, 86, 87, 103, 138, 172, 173].

In this chapter we deal with the $k$-connectivity augmentation problem. Generally speaking, given an undirected graph $G = (V, E_0)$ with $|V| = n$, and a feasible set $E$ of $m$ weighted edges on $V$ such that $G(V, E_0 \cup E)$ is $k$-edge (k-vertex) connected, the optimal $k$-edge (k-vertex) connectivity augmentation problem of $G = (V, E_0)$ is to find a subset $S^* \subseteq E$ such that $G(V, E_0 \cup S^*)$ is $k$-edge (k-vertex) connected and the weighted sum of the edges in $S^*$ is minimized.

If $G$ is an unweighted graph, and the edges in the feasible set $E = E(K_n) - E_0$ are unweighted, where $E(K_n)$ is the edge set of the complete graph $K_n$ on the vertex set $V$, Watanabe and Nakamura [172], Gabow [62], and Naor et al. [138] already showed that, for any $k < n$, the exact solution for the optimal $k$-edge connectivity augmentation problem can be obtained in polynomial time. When $k = 2$, Eswaran and Tarjan [46] gave the first linear algorithm for the optimal 2-connectivity augmentation problem. Rosenthal and Goldner [152] presented the first linear algorithm for the optimal biconnectivity augmentation problem. Later Hsu and Ramachandran [86] corrected an error in [152], and gave the first parallel algorithm for the problem. Their parallel algorithm requires $O(\log^2 n)$ time and $O(\max\{\frac{n \log n}{\log \log n}, \frac{m+n}{\log n}\})$ processors on an EREW PRAM. Hsu and Ramachandran [87], Watanabe and Nakamura [173] also presented algorithms for the optimal triconnectivity augmentation problem. Recently, Hsu [85] gave an algorithm for the optimal 4-vertex connectivity augmentation problem.
However, when the edges in the feasible set \( E \) are weighted, the situation is totally different. In this case, we cannot expect to find an exact solution \( S^* \) for the optimal \( k \)-connectivity augmentation problem in polynomial time even for \( k = 2 \). Eswaran and Tarjan [46] first showed that if \( G = (V, E_0) \) is disconnected, the optimal 2-connectivity augmentation problem is NP-complete. Frederickson and JáJá [56] further showed that even \( G = (V, E_0) \) is connected, this problem is still NP-complete [56]. Instead, Frederickson and JáJá [56] presented an \( O(n^2) \) time approximation algorithm for this problem, and the solution delivered by their algorithm is either twice the optimum if \( G \) is connected or 3 times optimum otherwise. Recently Khuller and Thurimella [103] presented another simple algorithm for the optimal 2-connectivity augmentation problem. Their algorithm requires \( O(m + n \log n) \) time, and the solution delivered is also 2 or 3 times optimum depending on \( G \) is connected or disconnected. However, if \( G \) is a special graph with its edge weights satisfying triangle inequality, there are better approximation algorithms for the optimal \( k \)-connectivity augmentation problem on this special graph. For example, Frederickson and JáJá [57] presented an approximation algorithm for the optimal 2-connectivity augmentation problem on it which delivers a solution within 1.5 times optimum.

A closely related problem is to find a minimum weighted/cardinality \( k \)-edge \((k\text{-vertex})\) connected subgraph in a \( k \)-edge \((k\text{-vertex})\) weighted/unweighted connected graph. This problem can be stated as follows. Given a \( k \)-edge \((k\text{-vertex})\) weighted or unweighted connected graph \( G(V, E) \) with \( k > 1 \), find a \( k \)-edge \((k\text{-vertex})\) connected subgraph \( G_1 = (V, E_1) \) such that \( G_1 \) has the minimum edge weight summation or the minimum edge cardinality, where \( E_1 \subseteq E \). By considering the case of \( E_0 = \emptyset \) and the case where the edges in \( E_0 \) have zero weight, it is easy to see that this problem is equivalent to the optimal \( k \)-connectivity augmentation problem. Obviously it is also NP-complete.

There are some approximation results for finding a minimum weighted/cardinality \( k \)-edge \((k\text{-vertex})\) connected subgraph on a \( k \)-edge \((k\text{-vertex})\) connected graph \( G \). When \( G \) is unweighted and \( k = 2 \), Khuller and Vishkin [104] presented the first \( O(m) \) time algorithms which deliver a solution for finding a minimum cardinality 2-edge connected subgraph within 1.5 times optimum, and a solution for finding a minimum cardinality biconnected subgraph within \( 5/3 \) times optimum. Later, by improving analysis of the Khuller and Vishkin algorithm, Garg et al. [69] showed the solutions by the algorithms of Khuller and Vishkin are respectively within 1.25 times optimum and 1.5 times optimum for these two problems. Recently Chong and Lam [25] developed NC approximation algorithms for the problems above using an approximating maximum cardinality matching technique [52]. The solution by their NC algorithm for finding a minimum cardinality 2-edge connected subgraph is \( 1.5 + \epsilon' \) times optimum, and the solution for finding a minimum cardinality biconnected subgraph is \( 7/4 + \epsilon' \) times optimum, where \( \epsilon' \) is an arbitrarily positive constant. As for general \( k \), Khuller and Raghavachari [102] recently obtained a solution for finding a minimum cardinality \( k \)-edge connected subgraph within 1.85 times optimum which is also the best result so far.

When \( G \) is weighted, to find an approximation solution for the minimum weighted
\( k \)-edge (\( k \)-vertex) connected subgraph problem seems more difficult. For \( k = 2 \) this special case, the approximation algorithm of Frederickson and Jákó for the optimal 2-connectivity augmentation problem \cite{Frederickson1987} implies a solution for this problem within 3 times optimum. Khuller and Vishkin \cite{Khuller1996} further improved the solution for finding a minimum weighted 2-edge connected subgraph, and presented a solution which is twice the optimum by applying the weighted matroid intersection algorithm of Gabow \cite{Gabow1992}. Very recently, Khuller and Raghavachari \cite{Khuller1999} obtained an approximation solution for finding a minimum biconnected subgraph within \( 2 + \frac{1}{n} \) times optimum by applying the submodular flow technique of Frank and Tardos \cite{Frank1984}. For general \( k \), Ravi and Williamson \cite{Ravi1998} presented an approximation solution for the problem of finding a minimum weighted \( k \)-vertex connected subgraph which is \( 2^\mathcal{H}_k \) times optimum, where 
\[
\mathcal{H}_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}.
\]

In this chapter, we will focus on the optimal \( k \)-connectivity augmentation problem on a weighted graph with fixed \( k \) by presenting parallel approximation algorithms for this problem. Our approach is to reduce the optimal \( k \)-connectivity augmentation problem to the minimum weighted set cover (MWSC for short) problem. Our contributions can be described as follows. By applying the NC approximation algorithm for the MWSC problem due to Berger et al. \cite{Berger1999} (here we mention that there is another simpler algorithm by Rajagopalan and Vazirani \cite{Rajagopalan1996} for this problem), we devise (i) an NC approximation algorithm for the optimal 2-edge connectivity augmentation problem which delivers a solution either within \((1 + \ln n_c)(1 + \epsilon)\) times optimum if \( G \) is connected, or within \((1 + \ln n_c)(1 + \epsilon) + 1\) times optimum otherwise; (ii) and an NC approximation algorithm for the optimal biconnectivity augmentation problem which delivers a solution within \((1 + \ln n_b)(1 + \epsilon)\log n_b\) times optimum if \( G \) is connected, or within \((1 + \ln n_b)(1 + \epsilon)\log n_b + 1\) times optimum otherwise, where \( n_c \) and \( n_b \) are the number of 2-edge connected components (2ECCs) and biconnected components (2VCCs) of \( G(V, E_0) \) respectively, and \( \epsilon \) is a small constant with \( 0 < \epsilon < 1 \). We also present an NC algorithm for the optimal \( k \)-edge connectivity augmentation problem with fixed \( k \) (> 2). For this latter case, if \( G = (V, E_0) \) is \((k - 1)\)-edge connected, our NC algorithm delivers an approximation solution which is either \( \beta t \) times optimum if \( k \) is odd or \( \beta t \) times optimum otherwise, where \( \beta = (1 + \ln n)(1 + \epsilon) \), \( 1 \leq t \leq \lceil \log \frac{1}{\epsilon} n \rceil \), and \( \epsilon \) is a constant with \( 0 < \epsilon < 1 \). If \( G \) is \( \lambda \)-edge connected and \( 1 \leq \lambda < k - 1 \), our NC approximation algorithm delivers a solution within \( \alpha \beta t \) times optimum, where \( \beta = (1 + \ln n)(1 + \epsilon) \), \( \epsilon \) is a constant with \( 0 < \epsilon < 1 \), and \( \alpha \) is defined as follows: (i) if both \( k \) and \( \lambda \) are odd (even), then \( \alpha = \frac{k - \lambda}{2}(t + 1) \); (ii) if \( k \) is even and \( \lambda \) is odd, then \( \alpha = t\frac{k - \lambda}{2} + \lfloor \frac{k - \lambda}{2} \rfloor \); (iii) otherwise, \( \alpha = t\frac{k - \lambda}{2} + \lfloor \frac{k - \lambda}{2} \rfloor + 1 \), where \( 1 \leq t \leq \lceil \log \frac{1}{\epsilon} n \rceil \). As a consequence of the results above, there exists an NC approximation algorithm for finding a minimum weighted \( k \)-edge connected subgraph on a \( k \)-edge connected weighted graph, delivering a solution at most \( \alpha \beta t \) times optimum, where \( \beta = (1 + \ln n)(1 + \epsilon) \), \( \epsilon \) is a constant with \( 0 < \epsilon < 1 \), and \( \alpha \) is defined as follows: \( \alpha = \frac{kt + k - t - 1}{2} \) if \( k \) is odd, otherwise \( \alpha = t\frac{k - 1}{2} + \lfloor \frac{k - 1}{2} \rfloor + 1 \), where \( 1 \leq t \leq \lceil \log \frac{1}{\epsilon} n \rceil \). The proposed NC algorithms for the problems above are the first NC algorithms giving these degrees of approximation. All previously known sequential algorithms for these problems seem hard to parallelize, therefore it is unlikely to derive an NC approximation algorithm.
from these previously known sequential algorithms.

The rest of this chapter is organized as follows. In Section 5.2, some basic concepts are introduced. In Section 5.3 we show how to augment a graph \( G \) into a 2-edge connected graph at optimal cost by presenting an NC approximation algorithm for it. In Section 5.4 we show how to augment a graph \( G \) into a biconnected graph at optimal cost by presenting an NC approximation algorithm for it. In Section 5.5 we first show how to augment a \((k - 1)\)-edge connected graph \( G \) to a \( k \)-edge connected graph at optimal cost by presented an NC approximation algorithm. Then we extend our technique for this special case to the general case where \( G \) has edge connectivity \( \lambda \), where \( 1 \leq \lambda < k \) and \( k \) is fixed.

5.2 Preliminaries

Let \( G_1 = (V, E) \) be an undirected, connected graph, perhaps with weights on the edges. The block tree, denoted by \( T(V \cup V_b, E_T) \), of \( G_1 \) is a graph which is defined as follows. \( V_b \) is the set of all 2VCCs of \( G_1 \). Let \( b \in V_b \) and \( v \in V \), there is an edge \((v, b)\) in \( E_T \) if \( v \) is in \( b \). It is well known that \( T \) is a tree. Let \( v \) be a vertex of \( T \), if \( v \in V \), \( v \) is a square vertex; otherwise \( v \in V_b \), \( v \) is a round vertex. It is clear that a square vertex whose degree is more than one in \( T \) is an articulation point of \( G_1 \).

Let \( T(V, E_T) \) be a rooted tree and \( Z \subset V \) with \( Z \neq \emptyset \), the vertex LCA(\( Z \)) of \( T \) is defined as follows: if \( Z = \{ v \} \), then LCA(\( Z \)) := \( v \); if \( Z = \{ u, v \} \), then LCA(\( Z \)) := LCA(\( u, v \)), i.e., the lowest common ancestor of \( u \) and \( v \) in \( T \); otherwise, LCA(\( Z \)) := LCA(\( Z - \{ u, v \} \cup \{ \text{LCA}(u, v) \} \)). A tree edge \( e \) in \( T \) is covered by a non-tree edge \( e' \in E \) if \( e \) is on the cycle consisting of \( e' \) and the tree edges.

It is well known that there are at most \( O(n^2) \) minimum edge cuts in a connected graph \( G_1 \) with \( n \) vertices. A compact way of representing these cuts was suggested by Dinic et al [39]. They call the representation as the cactus of \( G_1 \), denoted by \( \mathcal{H} \). Suppose \( G_1 \) has the minimum edge cut with the value \( \lambda > 0 \). Define the graph \( \mathcal{H} \) whose vertices are the classes of the equivalence relation \("v and w cannot be separated by a minimum edge cut"\). Two vertices of \( \mathcal{H} \) are joined by an edge if the two corresponding classes have an edge between them, and this edge of \( \mathcal{H} \) carries a weight equal to the number of such edges of \( G_1 \). The most important properties of \( \mathcal{H} \) are as follows.

**Lemma 5.1** [39]. (1) \( \mathcal{H} \) is a cactus, that is, all its 2VCCs are edges or simple cycles. For odd \( \lambda \), \( \mathcal{H} \) is in fact a tree.

(2) The edges of \( \mathcal{H} \) which are in cycles have weight \( \lambda/2 \) and the others have weight \( \lambda \).

(3) There is an exact correspondence between the minimum edge cuts in \( G_1 \) and the cuts in \( \mathcal{H} \) of total weight \( \lambda \).

Note that \( \mathcal{H} \) has at most \( n \) vertices and less than \( 3n/2 \) edges, and the cuts of \( \mathcal{H} \) defined in part (3) of Lemma 5.1 must consist either of a single edge of weight \( \lambda \), or of two edges of weight \( \lambda/2 \) in the same cycle.
An efficient sequential algorithm for constructing this representation was given by Karzanov and Timofeev [99], and an RNC algorithm was presented by Naor and Vazirani [137].

5.3 2-Edge Connectivity Augmentation

Let $G = (V, E_0)$ be connected and $E$ be a feasible set with $m$ weighted edges such that $G(V, E_0 \cup E)$ is 2-edge connected. We only need to show how to increase the edge connectivity of a tree due to the following facts. If $G$ has nontrivial 2ECCs, then we can contract the vertex sets of these components into single vertices, resulting in a tree whose edges are the bridges of $G(V, E_0)$. Let $E' \subseteq E$ be an edge set such that those edges in $E$ to be kept in $E'$ are the minimum weighted edges that connect the vertices of different 2ECCs of $G(V, E_0)$. It is easy to show that the computation of $E'$ can be finished in $O(\log n)$ time using $O(m)$ processors on an EREW PRAM provided all 2ECCs of $G$ are given. Note that the edges in $E - E'$ are useless for the augmentation.

From now on, we assume that the initial graph is a 2ECC tree $T$ rooted at $r$ with $n_c$ vertices where $r$ is a degree-one vertex and $n_c$ is the number of 2ECCs of $G$. A bipartite graph $B(V_1, V_2, E_b)$ is constructed as follows. $V_1$ denotes the set of the edges in $E'$ and $V_2$ denotes the edge set of $T$. There is an edge $(e_1, e_2) \in E_b$ and $e_1 \in V_i$, $i = 1, 2$, if and only if adding $e_1$ to $T$, the edge $e_2$ is on the cycle consisting of the tree edges and $e_1$. That is, $e_2$ is no longer a bridge after adding $e_1$ to $T$.

Lemma 5.2 The bipartite graph $B(V_1, V_2, E_b)$ defined can be constructed in $O(mn_c)$ time, where $|V_1| \leq m - n_c + 1$, $|V_2| \leq n_c$, and the weight of a vertex in $V_1$ is the weight of the corresponding edge in $G$.

Proof. First of all, we select a degree-one vertex as the root of $T$, and then traverse $T$ by assigning each vertex $v$ a pre-order numbering $\text{pre}(v)$ and the number of descendants (including itself) $\text{nd}(v)$. This assignment can be done in $O(\log n)$ time using $O(n)$ processors on an EREW PRAM. Now the construction of $B$ is as follows. Consider a non-tree edge $e_1 = (x, y)$ in $V_1$ and a tree edge $e_2 = (u, v)$ in $V_2$. Assume that $u$ is the parent of $v$ in $T$, there is an edge connecting vertices $e_1$ and $e_2$ in $B$ if and only if either one of the following conditions holds:

(i) $\text{pre}(v) \leq \text{pre}(x) < \text{pre}(v) + \text{nd}(v)$, and either $\text{pre}(y) < \text{pre}(v)$ or $\text{pre}(y) \geq \text{pre}(v) + \text{nd}(v)$;

(ii) $\text{pre}(v) \leq \text{pre}(y) < \text{pre}(v) + \text{nd}(v)$, and either $\text{pre}(x) < \text{pre}(v)$ or $\text{pre}(x) \geq \text{pre}(v) + \text{nd}(v)$.

Therefore, the graph $B$ can be constructed in $O(mn_c)$ time provided $E'$, $T$, and the pre-order numbering and the number of descendants for each vertex in $T$ are given. □

Now we show the following important lemma.

Lemma 5.3 Let $G(V, E)$ be a connected undirected graph, and $T(V, E_T)$ be the 2ECC tree of $G$. Then $G$ is 2-edge connected if and only if $V_1 = E - E_T$ dominates $V_2 = E_T$ in $B$, where the graph $B(V_1, V_2, E_b)$ is induced by the tree $T$ and the edge set $E - E_T$ defined as above.
Proof. If \( G \) is 2-edge connected, \( V_1 \) must dominate \( V_2 \) in \( B \). Assume that \( V_1 \) can not dominate \( V_2 \), then there exists a vertex \( e_2 \in V_2 \) which is not dominated by any vertex in \( V_1 \). Thus, \( e_2 \) is not in any simple cycle formed by tree edges and non-edge tree edges, i.e., \( e_2 \) is a bridge of \( G \) which contradicts that any 2-edge connected graph does not contain any bridges.

We now prove another part. If \( V_1 \) dominates \( V_2 \) in \( B \), each edge in \( T \) is included in a simple cycle at least, which means that the remaining graph is still connected after deleting any edge from \( G \). In other words, \( G \) does not contain any bridges. \( G \) therefore is 2-edge connected. \( \Box \)

From Lemma 5.3, it is easily derive that, for any subset \( S \subseteq V_1 \), if \( S \) dominates \( V_2 \) in \( B \), then the corresponding edge set by \( S \) is a 2-edge connectivity augmentation of \( G \). Let \( w(S) \) be the weighted sum of all vertices in \( S \). If we can find such a \( S^* \subseteq V_1 \) with minimizing \( w(S^*) \), then \( S^* \) is an optimal 2-edge connectivity augmentation. For the simplicity of expression later, \( S^* \) is called the minimum weighted dominator set on \( B \). Therefore the optimal 2-edge connectivity problem now becomes to find \( S^* \). In the following we show that this problem is equivalent to an MWSC problem. Let \( X = V_2 \), and for each vertex \( v \in V_1 \), there is a corresponding set \( A_v = \{ u \mid (u, v) \in E_0, v \in V_1, u \in V_2 \} \), and the weight of \( A_v \) is the weight of the corresponding edge of \( v \). Then to find \( S^* \) on \( B \) becomes to find a collection of sets \( A_v \) such that \( \cup A_v = X \) and the weighted sum of these sets is minimized. Obviously this is an MWSC problem for which Berger et al. [17] have shown the following theorem.

**Theorem 5.1** [17] Let \( H = (V, E) \) be a hypergraph with \( |V| = n' \) and \( |E| = m' \). For any \( 0 < \epsilon < 1 \), there is an \( NC \) algorithm for the minimum weighted set cover problem that uses \( O(m' + n') \) processors, runs in \( O(\log^{4} n' \log n' \log_{2}(n'n')/\epsilon^{5}) \) time, and produces a cover of weight at most \( (1 + \epsilon)(1 + \ln \Delta)\tau^{*} \), where \( \Delta \) is the maximum vertex degree and \( \tau^{*} \) is the optimal solution.

By Theorem 5.1, we know that there has been an \( NC \) approximation algorithm for the MWSC problem. If we can prove that the other steps of the proposed algorithm can be implemented in \( NC \), we thus obtain an \( NC \) approximation algorithm for the optimal 2-edge connectivity augmentation problem. Recall that our approximation algorithm for this problem consists of three stages. In the first stage it generates the 2ECC tree \( T \) if \( G \) is connected. Otherwise, adding the edges in the feasible set \( E' \) to \( G \) yields an MST, and then adding the tree edges to \( G \) produces the tree \( T \). In the second stage, it constructs a bipartite graph \( B \). In the third stage it finds an approximation solution of the minimum weighted dominator set on \( B \) which is equivalent to finding an approximation solution for the corresponding MWSC problem.

In the following we give the parallel implementation details for the three stages above. First of all, we show how to construct the 2ECC tree \( T \). Given a graph \( G = (V, E_0) \), to find all 2ECCs and bridges of \( G \) can be done by applying the biconnectivity algorithm of Tarjan and Vishkin [164]. That is, after finding all 2VCCs, we identify those 2VCCs consisting of one edge only which are bridges of \( G \), compute all CCs of the remaining graph by deleting all bridges from \( G \), construct a tree \( T \) whose
vertices are those non-bridge CCs, and whose edges are those bridges. If \( G \) is disconnected, we obtain a forest \( F \) rather than a tree \( T \). For this latter case, we add the edges in \( E' \) to \( G \), produce an MST by the algorithm of Chin et al. [22], and yield the tree \( T \) by adding the edges of the MST to \( F \).

In order to construct \( B \), we only need to check whether or not the two conditions in the proof of Lemma 5.2 hold. This can done easily provided the pre-order numbering and the number of descendents of each vertex in \( T \) are given. Note that the maximum vertex degree of \( B \) is \( n_c \). Having the graph \( B \), we obtain an approximation solution of the minimum weighted dominating set \( S^* \) on \( B \) by applying the NC algorithm for the MWSC problem due to Berger et al. [17]. We thus have

**Theorem 5.2** Given a weighted, connected graph \( G = (V, E_0) \) and a feasible set \( E' \), there exists an NC approximation algorithm for the optimal 2-edge connectivity augmentation problem which delivers a solution within \( (1 + \ln n_c)(1 + \epsilon) \) times optimum. The proposed algorithm requires \( O(\log^2 n/\epsilon^6) \) time and \( O(mn_c) \) processors on a CRCW PRAM, where \( n_c \) is the number of 2ECCs in \( G \) and \( \epsilon \) is a small constant with \( 0 < \epsilon < 1 \).

**Proof.** Now we analyze the computation complexity of the proposed NC approximation algorithm. The construction of \( T \) can be done in \( O(\log n) \) time using \( O(m + n) \) processors on a CRCW PRAM by the biconnectivity algorithm of Tarjan and Vishkin. The assignment of pre-ordering numbering and the number of descendents to each vertex in \( T \) can be done in \( O(\log n) \) time using \( O(n) \) processors [98]. The construction of \( B \) can be done in \( O(1) \) time using \( O(mn_c) \) processors on an EREW PRAM. Finding an approximation solution for the MWSC problem induced by \( B \) can be done in \( O(\log^2 n/\epsilon^6) \) time using \( O(mn_c) \) processors on a CRCW PRAM because \(|E_0| \leq mn_c\). The solution generated is within \((1 + \ln n_c)(1 + \epsilon) \) times optimum by Theorem 5.1, where \( n_c \) is the number of 2ECCs of \( G(V, E_0) \) and \( \epsilon \) is a small constant with \( 0 < \epsilon < 1 \).

When \( G = (V, E_0) \) is disconnected, find an MST of \( G(V, E_0 \cup E) \) by assigning the edges in \( E_0 \) with weight 0 and the edges in \( E \) with their original weight, add the edges of the tree to \( G(V, E_0) \), and generate a tree \( T \) defined as before. In the following we show that this leads to an approximation solution within \((1 + \ln n_c)(1 + \epsilon) + 1 \) times optimum where \( n_c \) is the number of 2ECCs of \( G(V, E_0) \) and \( \epsilon \) is a small constant with \( 0 < \epsilon < 1 \).

Let \( G^* \) be the minimum weighted 2-edge connected graph by optimal augmentation to \( G \), and \( w(G^*) \) be the associated weight of \( G^* \). The proof is proceeded as follows. We add all edges in \( G^* \) to \( T \), then all edges in \( G^* - T \) form a dominating set on \( B \) because \( G^* \) is 2-edge connected by Lemma 5.3. Therefore, the set by all edges in \( G^* - T \) dominates the edge set of \( T \). Let \( u(T^*) \) be the minimum weighted 2-edge augmentation on \( T \) such that the resulting graph is 2-edge connected. Then

\[
w(T^*) \leq w(G^* - T) \leq w(G^*).
\]

Furthermore, we have
\[ w(T) \leq w(G^*) , \]

because an MST of a graph is the minimum weighted connected subgraph of the graph. Therefore, we have the following theorem.

**Theorem 5.3** Given a disconnected graph \( G = (V,E_0) \) and a feasible set \( E' \), there exists an NC approximation algorithm for the optimal 2-edge connectivity augmentation problem which requires \( O(\log^7 n/\epsilon^6) \) time and \( O(mn_c) \) processors on a CRCW PRAM. The solution delivered by this algorithm is \( (1 + \ln n_c)(1 + \epsilon) + 1 \) times optimum, where \( n_c \) is the number of 2ECCs of \( G(V,E_0) \) and \( \epsilon \) is a small constant with \( 0 < \epsilon < 1 \).

As a consequence of Theorem 5.3, we have the following corollary.

**Corollary 5.1** Given a weighted 2-edge connected graph \( G(V,E) \), finding a 2-edge connected subgraph such that the weight of the subgraph is \( (1 + \ln n)(1 + \epsilon) + 1 \) times the weight of the minimum weighted 2-edge connected subgraph can be done in \( O(\log^7 n/\epsilon^6) \) time using \( O(mn) \) processors on a CRCW PRAM, where \( \epsilon \) is a small constant with \( 0 < \epsilon < 1 \).

### 5.4 Biconnectivity Augmentation

Assume that \( G = (V,E_0) \) is connected. Our strategy for this problem is similar to that used in the previous section. That is, we first obtain a 2VCC tree \( T \) of graph \( G \). We then construct a bipartite graph \( B \) and find an approximation solution \( S_i' \) of the MWSC problem induced by \( B_i \), where \( B_i \) is obtained from \( B_{i-1} \) and \( S = \bigcup_{j=1}^{i-1} S_j' \), \( 0 \leq i \leq [\log |V_2|] - 1 \). Initially, \( B_0 = B \) and \( S = \emptyset \). Let \( E'' \) be the corresponding edge set of \( \bigcup_{i=0}^{[\log |V_2|] - 1} S_i' \). The graph formed by adding all edges in \( E'' \) to \( G \) is a biconnected graph.

Define the 2VCC tree \( T \) as follows. The vertex set of \( T \) is \( V_4 \cup V_6 \), where \( V_4 \) represents the set by all articulation points of \( G \) and \( V_6 \) represents the set by all 2VCCs of \( G \). The edge set \( E(T) \) of \( T \) consists of edge \((a_i, b_j)\), where \( a_i \in V_4 \) and \( b_j \in V_6 \). By this definition, for each vertex \( v \) of \( V \), \( v \) is a vertex of \( T \) if \( v \) is an articulation point of \( G \); \( v \) is in a \( b_j \) otherwise, where \( b_j \) is the unique 2VCC containing \( v \). In the following, by **superimposing** an edge \((x,y)\) of \( E \) on \( T \), we mean adding an edge between \( a_i \), \( b_j \) of \( E(T) \), where \( x \) is either an articulation point of \( G \) (here \( x = a_i \)) or \( x \) is included in the 2VCC \( a_i \), and \( y \) is either an articulation point of \( G \) (here \( y = b_j \)) or \( y \) is included in the 2VCC \( b_j \) respectively. If there are multiple edges between two vertices in \( T \), we just keep the edge with the minimum weight, and remove all the other edges. Let \( E' \) be the remaining edge set, then \( |E'| \leq |E| \leq m \). Then, we only consider adding some edges in \( E' \) to make \( G \) into a biconnected graph.

Next, define the bipartite graph \( B(V_1, V_2, E_b) \) as follows. \( V_1 \) denotes the set of all edges in \( E' \), and \( V_2 \) denotes the set consisting of all 2VCCs of \( G \). There is an edge \((v_1, v_2) \in E_b \) and \( v_i \in V_i \), \( i = 1, 2 \), if and only if, adding the corresponding edge \( e = (x,y) \) of \( v_1 \) to \( T \), the vertex \( v_2 \) is on the cycle consisting of the tree edges and \( e \).

**Lemma 5.4** Given the 2VCC tree \( T \) and \( E' \), the graph \( B(V_1, V_2, E_b) \) can be constructed in \( O(mn_b) \) time, where \( \sum_{i=1,2} |V_i| \leq m + n_b \), \( |V_2| \leq n_b \).
**Proof.** Given the 2VCC tree $T$, construct an auxiliary tree $T'$ such that the LCA query of two vertices in $T$ can be answered in $O(1)$ time. The construction of $T'$ can be done in $O(n)$ time by Schieber and Vishkin's algorithm [154]. Now we construct the graph $B$ as follows. Let the corresponding edge of vertex $v_1 \in V_1$ be $e = (x, y)$, and $t = \text{LCA}(x, y)$ be the lowest common ancestor of $x$ and $y$ in $T$. Then there is an edge between $v_1$ and $v_2 \in V_2$ if and only if (i) $\text{LCA}(x, v_2) = x$ and $\text{LCA}(v_2, y) = v_2$ when $t = x$; (ii) $\text{LCA}(x, v_2) = v_2$ and $\text{LCA}(v_2, y) = y$ when $t = y$; (iii) either $\text{LCA}(t, v_2) = t$ and $\text{LCA}(v_2, x) = v_2$, or $\text{LCA}(t, v_2) = t$ and $\text{LCA}(v_2, y) = v_2$ when $t \neq x$ and $t \neq y$. Obviously $B$ can be obtained in $O(|E_b|) = O(mn_k)$ time. □

Denote by $G_B[X \cup Y]$ an induced subgraph of $B(V_1, V_2, E_b)$ on the vertex set in $X \cup Y$, where $X \cup Y \subseteq V_1 \cup V_2$. We proceed by the following lemma which is very important to construct our algorithm. In fact, our proposed algorithm is based on this lemma.

**Lemma 5.5** Let a subset $S \subseteq V_1$ and $S$ dominate the set $V_2$ in $B$. Then the graph formed by adding the corresponding edges of vertices in $S$ to $G$ is biconnected if and only if $G_B[S \cup V_2]$ is connected.

**Proof.** Let $S \subseteq V_1$ and $S$ dominate $V_2$. Assume that $G$ is not biconnected. The proof consists of two parts. In the first part, we show that if $G_B[S \cup V_2]$ is disconnected, the graph formed by adding the edges in $S$ to $G$ is not biconnected. In the second part we show that if $G_B[S \cup V_2]$ is connected, the graph formed by adding the edges in $S$ to $G$ is biconnected.

We start by showing the first part. Assume that $G_B[S \cup V_2]$ is disconnected, and has $k$ CCs with $k > 1$. Let $A$ and $B$ be two CCs among the $k$ CCs, and $V(A)$ and $V(B)$ be the vertex sets in $A$ and $B$ respectively. Let $b(A) = V(A) \cap V_2$ and $b(B) = V(B) \cap V_2$. Denote by $\alpha = \text{LCA}(b(A))$ and $\beta = \text{LCA}(b(B))$ on $T$. Then there exists a unique path $\pi_{\alpha\beta}$ between $\alpha$ and $\beta$ on $T$. Note that it is possible that $\pi_{\alpha\beta}$ consists of one vertex only. We further assume that $\pi_{\alpha\beta}$ contains no vertices belonging to other CCs except $A$ and $B$. The problem concerned now is divided into the following three cases:

(i) $\alpha \neq \beta$ and neither one is the ancestor of another in $T$. Then $\pi_{\alpha\beta}$ contains one more than vertices, and at least one vertex $v$ among these vertices is an articulation point of $G$ by the property of $T$, deleting $v$ will leave the vertices in $b(A)$ and the vertices in $b(B)$ in different CCs. Therefore, $v$ is still an articulation point of the graph formed by adding the edges in $S$ to $G$.

(ii) $\alpha = \beta$. In this case we further classify whether or not $\alpha$ is an articulation point of $G$. If it does, then deletion of $\alpha$ will leave the vertices in $b(A)$ and the vertices in $b(B)$ in different CCs again. Therefore, $\alpha$ is still an articulation point of the graph formed by adding the edges of $S$ to $G$. Otherwise, $\alpha$ is a 2VCC vertex. We show this is impossible. If $\alpha$ is a 2VCC vertex, it must be included in $V(A)$. By the same reason, it must be included in $V(B)$ also, then $A$ and $B$ should be the same CC rather than two distinguished CCs. It contradicts our initial assumption. Therefore, $\alpha$ is not a 2VCC vertex.
(iii) \( \alpha \neq \beta \) and one is the ancestor of another in \( T \). Here we assume that \( \beta \) is the ancestor of \( \alpha \). Let \( T_\alpha \) be a subtree of \( T \) rooted at \( \alpha \) including all vertices in \( b(A) \). By the same argument as case (ii), we can show that \( \alpha \) is an articulation point of \( G \) only. Meanwhile, we also note that there is no any edges between a vertex rather than \( \alpha \) in \( T_\alpha \) and a vertex in \( V_\alpha \cup V_\beta - V(T_\alpha) \) except the edges incident to \( \alpha \), which means that the deletion of \( \alpha \) will leave the vertices in \( T_\alpha \) and the other vertices of \( T \) separated, i.e., they are not in the same CC. Therefore, the graph formed by adding the edges in \( S \) to \( G \) is not biconnected.

Now we show the second part. Our approach is to show that, after adding the edges of \( S \) to \( G \), every articulation point of \( G \) is no longer an articulation point of the resulting graph.

Let \( v \) be an arbitrary articulation point of \( G \), and \( v \) be contained in \( l \) 2VCCs \( b_1, b_2, \ldots, b_l \). Then \( v \) is an adjacent vertex of these \( l \) vertices in \( T \). We need to prove that, if \( G_B[S \cup V_2] \) is connected, then all 2VCCs sharing \( v \) should become a 2VCC after adding all edges in \( S \) to \( G \). We start by finding all shortest paths between \( b_1 \) and \( b_j \) in \( G_B[S \cup V_2] \), where \( 2 \leq j \leq l \). Note that, there definitely exist these paths on \( G_B[S \cup V_2] \) because it is connected. Let the shortest path between \( b_1 \) and \( b_k \), denoted by \( P_{b_1,b_k} \), be the shortest among the \( l - 1 \) shortest paths, \( 2 \leq k \leq l \). Assuming that the vertex sequence of \( P_{b_1,b_k} \) is \( b_1, c_1, c_2, \ldots, c_p, b_k \), \( c_i \in V_1, c_j \in V_2 \), where \( 1 \leq i \leq p \), \( 1 \leq j \leq p - 1 \), and \( P_{b_1,b_k} \) does not contain any other \( b_j \) for \( j \neq k \).

If \( |P_{b_1,b_k}| = 1 \), by the definition of graph \( B \), \( b_1 \) and \( b_k \) are on the cycle by the tree edges of \( T \) and the edge \( e_1 \). We merge all 2VCCs on this cycle into a 2VCC. As a result, \( b_1 \) and \( b_k \) are merged into a 2VCC \( \ell \). Now \( v \) is still an articulation point of the augmented graph shared by \( l - 1 \) 2VCCs \( \ell', b_2, \ldots, b_{k-1}, b_{k+1}, \ldots, b_l \). We follow the above method to continue doing merge. Finally all initial 2VCCs sharing \( v \) are merged into one 2VCC, and \( v \) is no longer an articulation point of the graph formed by \( G \) and the edges in \( S \).

If \( |P_{b_1,b_k}| = p \) and \( p > 1 \). Then all vertices \( b_j \) for \( j \neq 1 \) and \( j \neq k \) do not appear on this path. By induction on \( p_i \), it is easy to prove that all 2VCCs on the path can be merged into one 2VCC. That means, after merging all 2VCCs on \( P_{b_1,b_k} \), \( b_1 \) and \( b_k \) are merged into a 2VCC \( \ell' \), and \( v \) now is the articulation point shared by \( l - 1 \) 2VCCs. We apply the above method again to merge all the remaining 2VCCs sharing \( v \). As a result, all \( b_i \) for \( 1 \leq i \leq l \) are merged into a 2VCC, and \( v \) is no longer an articulation point of the graph formed by \( G \) and the edges in \( S \).

Having Lemma 5.5, we assign each vertex in \( V_1 \) by the corresponding edge's weight. Let \( S^* \) be a \( S \) defined above with the minimum weight summation. Then the remaining task is to find such a \( S^* \). Obviously, this is an NP-complete problem again. Instead we look for an approximation solution for it. The basic idea of our approximation solution is to reduce this problem to a series of MWSC problems induced by \( B_i(V^{(i)}_1, V^{(i)}_2, E^{(i)}_b) \), \( 0 \leq i \leq \lfloor \log |V_2| \rfloor - 1 \).

Now we show how to construct the bipartite graph \( B_i(V^{(i)}_1, V^{(i)}_2, E^{(i)}_b) \) provided given \( B_{i-1} \) and a set \( S \subseteq V_1 \). Initially \( B_0(V^{(0)}_1, V^{(0)}_2, E^{(0)}_b) := B(V_1, V_2, E_b) \) and \( S := \emptyset \). We compute all CCs of graph \( G_B[S \cup V_2] \). Then a vertex \( v \in V^{(i-1)}_1 \) is included
in $V_1^{(i)}$ if and only if there exists at least two edges $(v, x), (v, y) \in E_0^{(i-1)}$ such that $x$ and $y$ are in different CCs of $G_B^*[S \cup V_2]$. $V_2^{(i)}$ is the set consisting of all CCs of $G_B^*[S \cup V_2]$. The edge set $E_0^{(i)}$ includes all edges such as $(v, c)$ and $(v, d)$, where $c$ is the CC containing $x$, $d$ is the CC containing $y$, $c \neq d$, and $(v, x), (v, y) \in E_0^{(i-1)}$. If there are one more edges between two vertices in graph $B_i$, we delete all other duplicate edges but one.

In the following we present an approximation algorithm for finding such a $S$ in Lemma 5.5.

$$S := \emptyset; V_1^{(0)} := V_1; V_2^{(0)} := V_2; E_0^{(0)} := E_0;$$

$B_0 := B(V_1^{(0)}, V_2^{(0)}, E_0^{(0)}); i := 0;$

while $G_B^*[S \cup V_2]$ is disconnected do

Find the minimum weighted dominator set $S_i$ on $B_i$;

$S := S \cup S_i$;

Compute all CCs of graph $G_B^*[S \cup V_2]$;

Construct the bipartite graph $B_{i+1}$;

$i := i + 1$
endwhile.

Note that the set $S_i$ in the algorithm cannot be obtained in polynomial time unless P=NP. However, we can give a polynomial approximation solution for it. Now the problem of finding a $S_i$ becomes the problem of finding an MWSC induced on $B_i$. Let $S_i'$ be an approximation solution of $S_i$ by the algorithm due to Berger et al. [17], then this solution is $(1 + \ln n_b)(1 + \epsilon)$ times optimum, where $n_b$ is the number of 2VCCs of $G(V, E_0)$ and $\epsilon$ is a small constant with $0 < \epsilon < 1$. Therefore, we have the following lemma.

**Lemma 5.6** Given $B$ is defined as above, let $S' \subseteq V_1$, $S'$ dominate $V_2$ and $G_B^*[S' \cup V_2]$ connected. Then we can find an approximation solution $S'$ which is $(1 + \ln n_b)(1 + \epsilon) \log n_b$ times optimum, where $n_b$ is the number of 2VCCs in a connected graph $G(V, E_0)$, i.e., $|V_2| = n_b$.

**Proof.** Assuming that $S^* \subseteq V_1$ has the minimum weight summation such that $S^*$ dominates $V_2$, and $G_B^*[S^* \cup V_2]$ is connected. From the above algorithm, it is obvious that $w(S_i) \leq w(S^*)$ because $S_i$ is such a vertex set with the minimum weight summation that dominates $V_2$, while the vertex set $S^*$, besides it must satisfy all properties of $S_i$, has an additional restriction that $G_B^*[S^* \cup V_2]$ is connected also. The other important observation is that the degree of each vertex $v_1 \in V_1$ is at least 2, i.e., there are at least two edges incident to $v_1$. Therefore, the number of vertices in $V_2$ is reduced at least a half from $B_i$ to $B_{i+1}$. However, $|V_2| \leq n_b$ initially. Thus, after $[\log n_b]$ iterations of the while loop, all vertices in $V_2$ are merged into a CC. The approximation solution obtained has the weight of $\sum_{i=0}^{[\log |V_2|]-1} w(S_i) \leq \sum_{i=0}^{[\log |V_2|]-1} (1 + \ln n_b)(1 + \epsilon)w(S_i) \leq [\log n_b](1 + \ln n_b)(1 + \epsilon)\max\{w(S_i)\} \leq [\log n_b](1 + \ln n_b)(1 + \epsilon)w(S^*).$
We now present the parallel implementation details for the optimal biconnectivity augmentation problem. The approach adopted is similar to that for the optimal 2-edge connectivity augmentation problem. The 2VCC tree $T$ is constructed as follows. Apply the biconnectivity algorithm of Tarjan and Vishkin [164] to find all 2VCCs on $G$, identify all articulation points of $G$. Note that a vertex is an articulation point if it appears in more than one 2VCCs. Then, construct the adjacency matrix of the 2VCC tree $T$, and run the algorithm for computing CCs of $T$ by Chin et al. [22] to establish the inverted tree $T$.

**Lemma 5.7** The 2VCC tree $T$ (stored as an inverted tree) can be constructed in $O(\log^2 n)$ time using $O(n^2/\log n)$ processors on a CRCW PRAM.

**Proof.** The algorithm for finding all 2VCCs requires $O(\log n)$ time and $O(m + n)$ processors on a CRCW PRAM [164]. The adjacency matrix of $T$ can be constructed in $O(\log n)$ time using $O(n^2)$ processors on an EREW PRAM. The inverted tree $T$ can be obtained in $O(\log^2 n)$ time using $O(n^2/\log n)$ processors on a CREW PRAM. □

The feasible set $E'$ can be generated in $O(\log n)$ time using $|E| \leq n^2$ processors on a CRCW PRAM. The details are as follows: assign the endpoints of every edge in $E$ with their labels (articulation points or 2VCC identifications) in $T$; sort these edges in ascending order by their endpoint labels as the first key and their associated weight as the second key; delete all other edges with the same labels but keep one with the minimum weight by applying prefix computation. So, the total computation can be finished in $O(\log n)$ time using $O(n^2)$ processors on a CRCW PRAM.

**Lemma 5.8** Given $T$, $T'$, and a feasible set $E'$, the graph $B$ can be constructed in $O(1)$ time using $O(mn_b)$ processors on an EREW PRAM where $n_b$ is the number of 2VCCs in $G$.

**Proof.** The vital step in the construction of $B$ is to check the three conditions in the proof of Lemma 5.4, which can be answered in $O(1)$ time provided $E'$, $T$, and $T'$ are given. Therefore, the construction of $B$ requires $O(1)$ time and $O(|E_b|) = O(mn_b)$ processors on an EREW PRAM. □

The rest is to find an approximation solution $S' \subseteq V_1$ of $B$ such that (i) $S'$ dominates $V_2$; and (ii) $G_B[S' \cup V_2]$ is connected; and (iii) $w(S') \leq [\log n_b](1 + \ln n_b)(1 + \epsilon)w(S^*)$, where $n_b$ is the number of 2VCCs of $G(V, E_0)$ and $\epsilon$ is a small constant with $0 < \epsilon < 1$. This $S'$ can be achieved by Lemma 5.6. We analyze the computational complexity of the proposed NC algorithm below.

The while loop can be checked in $O(\log n)$ time using $O(n/\log n)$ processors provided all CCs of $G_B[S \cup V_2]$ are given. This is implemented by using prefix computation to see whether all vertices in $V_2$ have the same CC labeling. The graph $B$ can be constructed in time $O(1)$ using $O(mn_b)$ processors on an EREW PRAM by Lemma 5.8. The step of finding an approximation solution $S'_i$ for $S_i$ in $B_i$ is more complicated. It consists of the following substeps: find an approximation solution for the MWSC
problem induced on $B_i$ by the algorithm due to Berger et al. [17], which requires $O(\log^7 n/e^6)$ time and $O(mn^y)$ processors on a CRCW PRAM, and the solution delivered is within $(1 + \ln n_b)(1 + e)$ times optimum [17], i.e., an approximation solution $S'_i$ with $w(S'_i) = (1 + \ln n_b)(1 + e)w(S_i)$ is obtained, where $e$ is a small constant with $0 < e < 1$; compute the CCs of $G_B[S \cup V_2]$ by Chong and Lams' algorithm [23], which requires $O(\log n \log \log n)$ time and $O(mn_b)$ processors. Having finished the steps above, the construction of $B_{i+1}$ can be done in $O(1)$ time using $O(mn_b)$ processors. Thus we have the following theorem.

**Theorem 5.4** Given a weighted, connected graph $G = (V, E_0)$ and a feasible set $E'$, there exists an NC approximation algorithm for the optimal biconnectivity augmentation problem which requires $O(\log^7 n \log n_b/e^6)$ time and $O(mn_b)$ processors on a CRCW PRAM. The solution delivered by the algorithm is $(1 + \ln n_b)(1 + e)\log n_b + 1$ times optimum, where $n_b$ is the number of 2VCCs in $G$ and $e$ is a small constant with $0 < e < 1$.

If $G$ is disconnected, we use the same method as in the previous section to make $G$ into a connected graph by adding the edges in the MST. Therefore,

**Theorem 5.5** Given a weighted, disconnected graph $G = (V, E_0)$ and a feasible set $E'$, there exists an NC approximation algorithm for the optimal biconnectivity augmentation problem which requires $O(\log^7 n \log n_b/e^6)$ time and $O(mn_b)$ processors on a CRCW PRAM. The solution delivered by the algorithm is $(1 + \ln n_b)(1 + e)\log n_b + 1$ times optimum, where $n_b$ is the number of 2VCCs in $G$ and $e$ is a small constant with $0 < e < 1$.

As a consequence of Theorem 5.5, we have the following corollary.

**Corollary 5.2** Given a weighted biconnected graph $G(V, E)$, finding a biconnected subgraph whose weight is $(1 + \ln n)(1 + e)\log n + 1$ times the weight of the minimum biconnected subgraph can be done in $O(\log^8 n/e^6)$ time using $O(mn)$ processors on a CRCW PRAM, where $e$ is a small constant with $0 < e < 1$.

### 5.5 $k$-Edge Connectivity Augmentation

#### 5.5.1 $k$-edge connectivity augmentation on a $(k - 1)$-edge connected graph

In this section we deal with the optimal $k$-edge connectivity augmentation problem on a $(k - 1)$-edge connected graph $G(V, E_0)$. Let $G = (V, E_0)$ be $(k - 1)$-edge connected, and $E$ be a feasible weighted edge set such that $G(V, E_0 \cup E)$ is $k$-edge connected, $m = |E|$. Assume that the cactus $\mathcal{H}$ of $G$ has already been built. Let $\mathcal{T}$ be the block tree of $\mathcal{H}$ which is also built, represented as an inverted tree rooted at a square node. To help the clarity of our description, we will refer to "nodes" of $\mathcal{H}$ and $\mathcal{T}$ but to "vertices" of other graphs.

By *superimposing* an edge $(x, y) \in E$ on $\mathcal{T}$, we mean adding an edge between the square nodes of $\mathcal{T}$ corresponding to the classes containing $x$ and $y$. If there are multiple
edges between the two nodes of \( \mathcal{T} \), we just keep the edge with the minimum weight, and remove the others. If we superimpose a sequence \( E' \) of edges on \( \mathcal{T} \), we will also refer to the consequent edges of \( \mathcal{T} \) as \( E' \). The original edges of \( \mathcal{T} \) will be called tree edges.

The following lemma is very important in the proof of the correctness of our proposed algorithm. In fact, the proposed algorithm is closely based on this lemma.

**Lemma 5.9** Assume that \( G \) is \((k - 1)\)-edge connected, \( \mathcal{H} \) is the cactus of \( G \), and \( \mathcal{T} \) is the block tree of \( \mathcal{H} \). If the augmented graph induced by adding the edges in \( S \subseteq E \) to \( G \) is \( k \)-edge connected, then all the tree edges of \( \mathcal{T} \) must be covered by the edges in \( S \).

**Proof.** Let \( G(V, E_0 \cup S) \) be the augmented graph induced by adding the edges in \( S \subseteq E \) to \( G \). We show that if there exists a tree edge \( e \) of \( \mathcal{T} \) which is not covered by any edge in \( S \), then \( G(V, E_0 \cup S) \) is not \( k \)-edge connected. Let \( e = \langle b, x \rangle \), where \( x \) is a square node and \( b \) is a round node. If \( e = \langle x, b \rangle \), the discussion is similar, and we are not going to discuss this latter case further. We proceed our discussion by the following two cases.

(i) If \( k - 1 \) is odd, then \( \mathcal{H} \) is a tree, and each 2VCC of \( \mathcal{H} \) is trivial, consisting of one edge only. Let \( \langle x, y \rangle \) be the edge of \( b \). We claim that if the edge \( e = \langle b, x \rangle \) of \( \mathcal{T} \) is not covered, the edge \( \langle b, y \rangle \) of \( \mathcal{T} \) is not covered either because all paths from \( x \) to \( y \) in \( \mathcal{H} \) go through the edge \( \langle x, y \rangle \) by the definition of this edge. Note that the endpoints of the edges in \( S \) are square nodes only. By the definition of \( \mathcal{H} \), deletion of the edge \( \langle x, y \rangle \) from \( \mathcal{H} \) corresponds to a partition of vertices in \( G \), which has a cut \( k - 1 \). That is, \( G(V, E_0 \cup S) \) — the resulting graph induced by adding the edges in \( S \) to \( G \) is still not \( k \)-edge connected if the tree edge \( e \) is not covered.

(ii) If \( k - 1 \) is even, then all minimum edge cuts of graph \( G \) are either tree-edge cuts or cycle-edge cuts, represented by either an edge in \( \mathcal{H} \) not belonging to any cycles or any pair of edges on a cycle of \( \mathcal{H} \). If \( e = \langle b, x \rangle \) and \( b \) consists of one edge only, then the edge of \( \mathcal{H} \) in \( b \) is a bridge of \( \mathcal{H} \), this becomes case (i), the lemma follows. Otherwise, \( b \) is not a trivial 2VCC, and all edges in \( b \) form a simple cycle \( C \) of \( \mathcal{H} \) by Lemma 5.1. Obviously, \( b \) contains at least three nodes. Let \( y \) and \( z \) be the adjacent nodes of \( x \) in \( C \). Then, the tree edge between \( b \) and node \( y \) (\( z \)) of \( \mathcal{T} \) must be in the form \( \langle y, b \rangle \) (\( \langle z, b \rangle \)), because \( x \) is already the parent of \( b \). Let \( \text{pre}(x) \) be the pre-order numbering of \( x \) in \( \mathcal{T} \), and \( \text{nd}(x) \) be the number of descendend nodes in the subtree rooted at \( x \) (the root itself is counted as one of its children). Then there does not exist an edge \( (s, t) \in S \) such that (1) \( \text{pre}(s) \leq \text{pre}(e) < \text{pre}(x) + \text{nd}(x) \); and (2) either \( \text{pre}(t) < \text{pre}(x) \) or \( \text{pre}(t) \geq \text{pre}(x) + \text{nd}(x) \). Otherwise, the edge \( e \) is covered by such an edge. We now show that deleting the edges \( \langle x, y \rangle \) and \( \langle x, z \rangle \) will lead to a partition of nodes in \( \mathcal{H} \), and therefore leads to a cut of \( G(V, E_0 \cup S) \). Because the edge \( e \) is not covered by any edge in \( S \), so, deleting the edges \( \langle x, y \rangle \) and \( \langle x, z \rangle \) from \( C \) leaves \( \mathcal{H} \) disconnected and it becomes at least two disconnected subgraphs, one including \( y \) (\( z \)), and the other including \( x \). That is, the minimum edge cut in the augmented graph \( G(V, E_0 \cup S) \) is still \( k - 1 \) rather than \( k \).

Lemma 5.9 suggests a necessary condition for \( k \)-edge connectivity augmentation.
However this condition is not sufficient. To illustrate that, we present a counterexample below. Let $G''$ be a 2-edge connected graph with 8 vertices. Assume that $G''$ is a simple cycle and the vertices on this cycle are numbered from 1 to 8 in clockwise. The objective is to augment $G''$ to 3-edge connected by a feasible weighted set $E$. Obviously, the cactus $\mathcal{H}$ of $G''$ is also a simple cycle which is the same as $G''$. Let $T$ be the block tree of $\mathcal{H}$, and $S = \{(1, 3), (2, 4), (5, 7), (6, 8)\} \subseteq E$ be an edge set. It is easy to see that all the tree edges in $T$ are covered by the edges in $S$. But the augmented graph induced by adding the edges in $S$ to $G''$ is not 3-edge connected. See Figure 5.1.

![Figure 5.1: An example.](image)

Having Lemma 5.9, we show how to augment a $(k-1)$-edge connected graph to a $k$-edge connected graph at optimal cost. Let $G_0 = G(V, E_0)$. Our approach is to augment $G_0$ by a sequence of augmentation, i.e., we yield a series of graphs $G_0, G_1, \ldots, G_t$, where $G_i$ is $k$-edge connected and $G_i$ is $(k-1)$-edge connected for all $i, i < t$. We later show that $t = O(\log n)$. The graph $G_{i+1}$ is generated by the following approach. Let $\mathcal{H}_i$ be the cactus of $G_i$, and $T_i$ be the block tree of $\mathcal{H}_i$, $0 \leq i \leq t$. $G_{i+1}$ is the augmented graph by adding the edges in $S_i$ to $G_i$, where $S_i \subseteq E$ is an edge set such that the edges in $S_i$ cover all tree edges of $T_i$ and $w(S_i)$ is minimized, $0 \leq i \leq t$.

**Lemma 5.10** Assume that $G_i$ and $T_i$ are defined as above. Let $w(S^*)$ be the optimal cost to augment a $(k-1)$-edge connected graph $G$ to a $k$-edge connected graph by adding the edges in $S^*$ into $G$, and $w(S_i^*)$ be the optimal cost to cover all tree edges of $T_i$ by the edges in $S_i^*$. Then $w(S_i^*) \leq w(S^*)$, $0 \leq i \leq t - 1$, where $S^* \subseteq E$ and $S_i^* \subseteq E$.

**Proof.** By Lemma 5.9, if the edges in $S_0^*$ cover all tree edges of $T_0$ and $G(V, E_0 \cup S_0^*)$ is $k$-edge connected, then obviously $w(S_0^*) = w(S^*)$; otherwise, $w(S_0^*) \leq w(S^*)$. Now we
already know that $G_1 = G(V, E_0 \cup S_0^*)$ is not $k$-edge connected. Then when adding the edges of $S^* - S^* \cap S_0^*$ to $T_1$, all tree edges in $T_1$ are covered, whereas $S_1^*$ is the edge set with the minimum weighted sum to cover all edges in $T_1$, so $w(S_1^*) \leq w(S^* - S^* \cap S_0^*) < w(S^*)$. If $G_2$ is $k$-edge connected, then the lemma follows. Otherwise we repeat this procedure until the final augmented graph is $k$-edge connected. □

**Lemma 5.11** Let $G_0 = G(V, E_0)$ be a $(k-1)$-edge connected graph, and $G_i$ be the first $k$-edge connected graph by a series of augmentation via yielding graphs $G_i$, $0 \leq i \leq t$. Then $1 \leq t \leq \lceil \log_2 n \rceil$.

**Proof.** Recall that $\mathcal{H}_0$ is the cactus of $G_0$, $S_0^*$ is a edge set whose edges cover all tree edges of $T_0$ with minimizing $w(S_0^*)$, and $\mathcal{H}_1$ is the cactus of the graph $G(V, E_0 \cup S_0^*)$. Let $\mathcal{H}_0'$ be the graph obtained by deleting all cycle-edges in $\mathcal{H}_0$. Then all nodes in a CC of $\mathcal{H}_0'$ now belong to a node of $\mathcal{H}_1$, i.e., these nodes are merged into a new node, which also means that the graph induced by the vertices of $G_0$ in the new node are $k$-edge connected. This approach can be generalized from $G_i$ to $G_{i+1}$. From this discussion, we know that those nodes which are not in any cycles are always merged into some cycle nodes. Therefore, in the following we only consider the nodes in the cycles of a cactus.

Let $\mathcal{H}_i'$ be the graph obtained by deleting all cycle-edges in $\mathcal{H}_i$, $\mathcal{H}_i^c$ be the compression representation of $\mathcal{H}_i$ by compressing all nodes in a CC of $\mathcal{H}_i'$ into a new node. Then $\mathcal{H}_i^c$ has the following properties: (i) $\mathcal{H}_i^c$ does not contain any tree-edge, i.e, $\mathcal{H}_i^c$ is 2-edge connected; (ii) For any two simple cycles in $\mathcal{H}_i^c$ they share a common node at most. And each simple cycle of $\mathcal{H}_i^c$ contains three nodes at least.

Let $n_i'$ be the number of nodes in $\mathcal{H}_i^c$, we show that $n_{i+1}' \leq \frac{3}{4}n_i'$. To prove that, consider the block tree $T_i^c$ of $\mathcal{H}_i^c$. Let $n_l$ be the number of leaf nodes of $T_i^c$, $n_c$ be the number of edges in $T_i^c$, $n_r$ be the number of round nodes in $T_i^c$, and $n_c$ be the number of non-leaf square nodes, i.e., the number of articulation points in $\mathcal{H}_i^c$. Then we have

$$n_c + n_l = n_i'$$  \hfill (5.1)

$$n_r \geq n_c$$  \hfill (5.2)

$$m_c = n_r + n_l + n_c - 1$$  \hfill (5.3)

Meanwhile, it is easy to derive that

$$m_c \geq \frac{3n_r + 2n_c + n_l}{2}$$  \hfill (5.4)

because the degree of each round node of $T_i^c$ is at least three by property (ii) of $\mathcal{H}_i^c$, and the degree of each articulation point is at least two.

By Equations 5.1 and 5.3, and Inequalities 5.2 and 5.4, we have

$$n_l \geq n_r + 2 \geq n_c + 2 = n_i' - n_l + 2$$
i.e.,

$$n_t \geq n'_t/2 + 1 > n'_t/2.$$  \hfill (5.5)

Since the edges in \(S_t^*\) cover all tree edges of \(T_t\), they cover all tree edges of \(T_t^c\). By the edges in \(S_t^*\), all leaf nodes of \(T_t^c\) which are also the nodes of \(H_t^c\) are merged with some other nodes. Therefore, the number of leaf nodes is reduced at least half after each merging. Thus, the number of nodes of \(H_t+1\) is \(n_{t+1}'\leq n'_t-n_t/2 > n'_t-n'_t/4 = \frac{3}{4} n'_t.\) Note that all those nodes that are not in any cycle of \(H_t^c\) are merged with some other nodes by the edges in \(S_t^*\). Therefore, after \(t = \lceil \log_4 n \rceil\) augmentations, all nodes of \(T_0\) are merged into a node, so the lemma follows. \(\Box\)

**Lemma 5.12** Let \(w(S^*)\) be the optimal cost to augment a \((k-1)\)-edge connected graph \(G\) to \(k\)-edge connected by adding the edges in \(S^*\) to \(G\) where \(S^* \subseteq E\), and \(w(S_t^*)\) be the optimal cost to cover all tree edges of \(T_t\) by the edges in \(S_t^* \subseteq E\). Then the cost used for augmenting \(G = G_0 \rightarrow G_t\) is at most \(w(S^*)t\), where \(1 \leq t \leq \lceil \log_4 n \rceil\).

**Proof.** Combined Lemmas 5.10 and 5.11, the total cost is \(\sum_{i=0}^{t-1} w(S_t^*) \leq w(S^*)t. \) \(\Box\)

When \(k - 1\) is odd, the cactus \(H\) of \(G\) is a tree, for this special case, it is unnecessary to yield \(G_t, 0 \leq i \leq t\). This is witnessed by the following lemma.

**Lemma 5.13** Let \(G\) be \((k-1)\)-edge connected, \(H\) be the cactus of \(G\), and \(T\) be the block tree of \(H\). If \(H\) is a tree, the augmented graph induced by adding all edges of \(S \subseteq E\) to \(G\) is \(k\)-edge connected if and only if all tree edges of \(T\) are covered by the edges in \(S\).

**Proof.** If the augmented graph is \(k\)-edge connected, then all tree edges must be covered by the edges in \(S\) which is already proven by Lemma 5.9. Now assume that \(H\) is a tree, we show that if all edges of \(T\) are covered by the edges in \(S\), the graph \(G(V, E_0 \cup S)\) is \(k\)-edge connected. We proceed by showing that there are \(k\) edge-disjoint paths on \(G(V, E_0 \cup S)\) for each pair of vertices \(u\) and \(v\). Recall that, by the definition of \(G\), there are \(k - 1\) edge-disjoint paths between \(u\) and \(v\) in \(G\). Let \(x\) and \(y\) in \(H\) be the nodes to which the vertices \(u\) and \(v\) of \(G\) belong respectively. Note that both \(x\) and \(y\) are square nodes in \(T\) and there is a unique path \(\pi_{x,y}\) between \(x\) and \(y\) because \(H\) is a tree. If all the edges on \(\pi_{x,y}\) are covered by an edge \(e \in S\), then all square nodes on \(\pi_{x,y}\) are merged into a new square node, i.e., all those vertices of \(G\) belonging to the nodes on \(\pi_{x,y}\) now belong to the new square node. As a result, the graph induced on those vertices of \(G\) in a new square node are \(k\)-edge connected. Therefore, there are \(k\) edge-disjoint paths between \(u\) and \(v\) after adding \(e\) to \(G\). Otherwise, all edges on \(\pi_{x,y}\) are covered by a non-tree edge sequence \(e_1, e_2, \ldots, e_l\), \(e_i \in S\) and \(1 \leq i \leq l\), for this case we can easily show that there are \(k\) edge-disjoint paths between \(u\) and \(v\) in the augmented graph after adding \(e_1, e_2, \ldots, e_l\) to \(G\) by induction on \(l. \) \(\Box\)
By the discussion above, now it is clear that the optimal $k$-edge connectivity augmentation problem of $G$ can be reduced to find a series of edge sets with minimizing weighted sum such that the edges in each such edge set cover all tree edges of the corresponding block tree.

From now on, assume that the block tree $T$ of a graph with at most $O(n)$ vertices is given, we show how to find an edge set $S \subseteq E$ to cover all edges of $T$ such that $w(S)$ is minimized. We select a vertex $r$ with degree one in $T$ as the root of this tree. Then we construct a bipartite graph $B(V_1, V_2, E_b)$ as follows. $V_1$ denotes the set of all edges of $E$, and $V_2$ denotes the edge set of $T$. There is an edge $(e_1, e_2) \in E_b$ if $e_i \in V_i$, $i = 1, 2$, if and only if by adding $e_1$ to $T$, the edge $e_2$ is on the cycle consisting of the tree edges and $e_1$. That is, $e_2$ is no longer a bridge of $T$ after adding $e_1$ to $T$.

**Lemma 5.14** The bipartite graph $B(V_1, V_2, E_b)$ defined can be constructed in $O(nm)$ time, where $|V_1| \leq m$ and $|V_2| \leq c'n - 1$ and the weight for each vertex in $V_1$ is the weight of corresponding edge of $G$, where $c_n, c'n$ are the number of vertices of $H$ and $T$ respectively, and $c' \geq c$ is a constant.

**Proof.** The proof is similar to that used in Lemma 5.2, omitted. □

Obviously for any subset $S \subseteq V_1$, if $S$ dominates $V_2$, the edges in the corresponding edge set of $S$ cover all edges in $T$. That is, if we can find an edge set $S^* \subseteq V_1$ such that $S^*$ dominates $V_2$ and $w(S^*)$ is minimized, then $S^*$ is the solution of the optimal cover of all edges of $T$. However, there is no way to find such a $S^*$ in polynomial time unless $P = NP$ because it is an MWSC problem. Instead there is an NC approximation algorithm for the MWSC problem due to Berger et al. [17]. We thus have

**Theorem 5.6** Given a $(k - 1)$-edge connected graph $G = (V, E_0)$ and a feasible weighted edge set $E$ such that $G(V, E_0 \cup E)$ is $k$-edge connected, there exists an approximation algorithm for the optimal $k$-edge connectivity augmentation problem which delivers a solution within either $\beta t$ times optimum if $k$ is odd or $\beta$ times optimum otherwise, where $\beta = (1 + \ln n)(1 + \epsilon)$, $1 \leq t \leq \lceil \log_\frac{1}{\epsilon} n \rceil$ and $\epsilon$ is a constant with $0 < \epsilon < 1$.

**Remarks.** Though the above approximation algorithm is inefficient in the sequential environment, an NC approximation algorithm with the same degree of approximation is derived from it, but all previously known sequential algorithms for this problem seem hard to parallelize.

Recall that our approximation algorithm for the optimal $k$-edge connectivity augmentation problem consists of $t$ stages, where $1 \leq t \leq \lceil \log_\frac{1}{\epsilon} n \rceil$. In stage $i$ there are three steps. Step 1 generates the cactus $H_i$ by graph $G_i$ and the block tree $T_i$ of $H_i$. Step 2 constructs a bipartite graph $B_i$. Step 3 finds an approximation solution for the MWSC problem induced on $B_i$ which is a part of the final approximation solution of the original problem.

In the following we give the implementation details with respect to the three steps in each stage. We start by constructing the cactus $H_i$ of $G_i$. For general $k$, there is
an RNC algorithm by Naor and Vazirani [137] for it. The only randomization part in their algorithm is a subroutine for computing the minimum edge cut between a pair of vertices. In this chapter we assume that $k$ is fixed. For this special case, the graph $\mathcal{H}_i$ can be built in NC. What we need to do is to replace that subroutine by the algorithm of Khuller and Schieber [101] which is in NC. So, we have

**Lemma 5.15** Given a $(k-1)$-edge connected graph $G_i = (V, E_0 \cup \sum S_j)$ with fixed $k$, the cactus $\mathcal{H}_i$ of $G_i$ can be built in $O(k^2 \log n)$ time using $O(n^3)$ processors on a CRCW PRAM, where $S_i$ is the edge set used to augment $G_i$ to $G_{i+1}$ which is derived by applying the above algorithm.

**Proof.** First of all, we find a sparse $k$-edge certificate $SC_i$ of $G_i$ which can be done in $O(k \log n)$ time using $C(n, m_0 + kn)$ processors by the algorithm of Cheriyan et al. [21], where $C(n, m)$ is the number of processors to compute all CCs of a graph with $n$ vertices and $m$ edges in $O(\log n)$ time on a CRCW PRAM. Note that the cactus $\mathcal{H}_i^{SC}$ of $SC_i$ is the cactus $\mathcal{H}_i$ of $G_i$ by the property of the certificate. Then combining Khuller and Schieber's algorithm with Naor and Vazirani's algorithm, the graph $\mathcal{H}_i$ can be built in $O(k^2 \log n)$ time using $O(\max\{nC(n, kn), n^3\}) = O(n^3)$ processors on a CRCW PRAM. \(\square\)

Having $\mathcal{H}_i$, the block tree $T_i$ of $\mathcal{H}_i$ can be built easily by applying the biconnectivity algorithm of Tarjan and Vishkin [164]. That is, after finding all 2VCCs of $\mathcal{H}_i$, the construction of $T_i$ is straightforward.

We now turn to the step 2. It is not difficult to show that $B_i$ can be constructed in $O(\log n)$ time using $O(mn)$ processors by Lemma 5.2. Having the graph $B_i$, find an approximation solution for the MWSC problem induced on $B_i$ by applying the NC algorithm of Berger et al [17]. The solution delivered is $(1 + \ln n)(1 + \epsilon)$ times optimum, where $\epsilon$ is a constant with $0 < \epsilon < 1$. Therefore we have

**Theorem 5.7** Given a $(k-1)$-edge connected graph $G = (V, E_0)$ and a feasible weighted edge set $E$ such that $G(V, E_0 \cup E)$ is $k$-edge connected, there exists an NC approximation algorithm for the optimal $k$-edge connectivity augmentation problem which delivers a solution within $\beta t$ times optimum if $k$ is odd, or within $\beta t$ times optimum otherwise, and the proposed algorithm requires $O(\log^8 n/\epsilon^6)$ time and $O(n^\beta)$ processors on a CRCW PRAM, where $1 \leq t \leq \lceil \log_8 n \rceil$, $\beta = (1 + \ln n)(1 + \epsilon)$, and $\epsilon$ is a constant with $0 < \epsilon < 1$.

**Proof.** The correctness of the algorithm is already shown by Lemmas 5.9, 5.10, 5.11 and 5.12. Now we analyze its parallel implementation complexity. The construction of $\mathcal{H}_i$ requires $O(k^2 \log n)$ and $O(\max\{nC(n, kn), n^3\})$ processors. The construction of $T_i$ can be done in $O(\log n)$ time using $O(n)$ processors on a CRCW PRAM by the biconnectivity algorithm of Tarjan and Vishkin, because $\mathcal{H}_i$ contains only $O(n)$ nodes and $O(n)$ edges. The assignment of pre-order numbering and the number of descendents for every vertex can be done in $O(\log n)$ time using $O(n)$ processors [98]. The construction of $B$ can be done in $O(1)$ time using $O(mn)$ processors on an EREW
PRAM. $B_k$ can be constructed in $O(\log n)$ time using $O(mn)$ processors on an EREW PRAM. Finding an approximation solution for the MWSC problem induced on $B_k$ can be finished in $O(\log^2 n/e^2)$ time using $O(mn)$ processors on a CRCW PRAM because $|E_k| \leq c^m$, and the solution generated is $\beta$ times optimum, this leads to an approximation solution for the original problem which is within $\beta t$ times optimum, where $1 \leq t \leq \lceil \log_2 n \rceil$, $\beta = (1 + \ln n)(1 + \epsilon)$, and $\epsilon$ is a small constant with $0 < \epsilon < 1$. Particularly, when $k$ is even, the solution is within $\beta$ times optimum. □

5.5.2 $k$-edge connectivity augmentation on a general graph

In the following we show how to augment a $\lambda$-edge connected graph $G = (V, E_0)$ to $k$-edge connected optimally by a given feasible weighted edge set $E$ with $G(V, E_0 \cup E)$ is $k$-edge connected, $1 \leq \lambda < k - 1$. We start by the following two lemmas.

**Lemma 5.16** Let $A_i^*$ be the $i$-edge connected graph by optimal augmentation of $G$ using the feasible weighted edge set $E$. Then $w(A_i^*) \leq w(A_j^*)$ for all $i$ and $j$ if $i < j$ provided $G(V, E_0 \cup E)$ is $j$-edge connected at least.

The proof of this lemma is so easy, we omit it here.

**Lemma 5.17** Let $A_i$ be an $i$-edge connected subgraph of $G(V, E_0 \cup E)$, and let $S_i^* \subseteq E$ be the edge set by optimal augmenting $A_i$ to $(i + 1)$-edge connected. If $i$ is odd, then $w(S_i^*) \leq w(A_{i+1}^*)$, otherwise $w(S_i^*) \leq w(A_{i+1}^*)[\log_2 n]$.

**Proof.** Combining Lemmas 5.10 and 5.11, the lemma then follows. □

From the lemmas above we have the following theorem.

**Theorem 5.8** Given a $\lambda$-edge connected graph $G = (V, E_0)$ with $1 \leq \lambda < k - 1$, a feasible weighted edge set $E$ such that $G(V, E_0 \cup E)$ is $k$-edge connected, there exists an NC approximation algorithm for the optimal augmenting $G$ to $k$-edge connected. This algorithm delivers a solution within $\alpha \beta$ times optimum, where $\beta = (1 + \ln n)(1 + \epsilon)$, $\epsilon$ is a constant with $0 < \epsilon < 1$, and $\alpha$ is defined as follows: (i) if both $k$ and $\lambda$ are odd (even), then $\alpha = \left\lfloor \frac{k-\lambda}{2} \right\rfloor (t + 1)$; (ii) if $k$ is even and $\lambda$ is odd, then $\alpha = t\left[\left\lfloor \frac{k-\lambda}{2} \right\rfloor + \left\lceil \frac{k-\lambda}{2} \right\rceil \right]$; (iii) otherwise, $\alpha = t\left[\left\lfloor \frac{k-\lambda}{2} \right\rfloor + \left\lceil \frac{k-\lambda}{2} \right\rceil \right]$, where $1 \leq t \leq \lceil \log_2 n \rceil$.

**Proof.** Because $G$ is $\lambda$-edge connected, we augment $G$ by a sequence of augmentation. That is, first we augment $G$ to $(\lambda + 1)$-edge connected by the algorithm in Section 5.5.1. As a result, we obtain a $(\lambda + 1)$-edge connected augmented graph. Then we augment this augmented graph to $(\lambda + 2)$-edge connected, and so on, until we obtain the final graph, denoted by $G^{(k)}$, which is $k$-edge connected. In the following we only prove case (i). The other cases are similar to this case, omitted. When both $k$ and $\lambda$ are even, by Theorem 5.7, Lemmas 5.16 and 5.17, we have

$$w(G^{(k)}) \leq w(A_\lambda) + \beta(t(w(S_\lambda^*) + (w(S_{\lambda+2}^*) \ldots + w(S_{k-2}^*)))$$

$$+ \beta(w(S_{\lambda+1}^*) + w(S_{\lambda+3}^*) \ldots + w(S_{k-1}^*))$$

$$\leq \beta(k - \lambda)/2(t + 1)w(A_k^*).$$
where \( w(A_\lambda) = 0, \beta = (1 + \ln n)(1 + \epsilon) \) and \( \epsilon \) is a constant with \( 0 < \epsilon < 1 \). □

If the graph \( G = (V, E_0) \) is disconnected, find an MST in graph \( G(V, E_0 \cup E) \) by assigning the edges in \( E_0 \) with weight zeros and the edges in \( E \) with their original weights, add the edges of the tree to \( G(V, E_0) \), and generate a 1-edge connected graph \( A_1 \). Let \( E_{mst} \) be the edge set in \( E \) whose edges are included in the MST. Obviously, \( w(E_{mst}) \leq w(A_i^*) \) for all \( i \geq 1 \). Note that an MST can be found by the NC algorithm due to Chin et al [22]. In this case, if \( k \) is odd, then \( w(G^{[k]}) \leq \beta^{\frac{k+1}{2} - \frac{k-1}{2}} w(A_k^*) \); otherwise \( w(G^{[k]}) \leq \beta(t^\left[\frac{k-1}{2}\right] + \left[\frac{k-1}{2}\right] + 1)w(A_k^*) \), where \( 1 \leq t \leq \lfloor \log_2^\frac{1}{n} \rfloor \), \( \beta = (1 + \ln n)(1 + \epsilon) \), and \( \epsilon \) is a constant with \( 0 < \epsilon < 1 \). Consequently we have

**Theorem 5.9** Given a weighted \( k \)-edge connected graph \( G(V, E) \), there exists an NC approximation algorithm for finding a minimum \( k \)-edge connected subgraph which delivers a solution within \( \alpha \beta \) times optimum, where \( \beta = (1 + \ln n)(1 + \epsilon) \), \( \epsilon \) is a constant with \( 0 < \epsilon < 1 \), and \( \alpha \) is defined as follows: \( \alpha = \frac{k+1}{k-2}, k \) is odd, \( \alpha = t^\left[\frac{k-1}{2}\right] + \left[\frac{k-1}{2}\right] + 1 \) otherwise, where \( 1 \leq t \leq \lfloor \log_2^\frac{1}{n} \rfloor \).

**Remarks.** Regarding finding the minimum \( k \)-edge connected subgraph problem, there still exists a big gap between our parallel solution within \( \alpha \beta \) times optimum and the sequential solution within twice the optimum by Khuller and Vishkin [104]. However, it seems unlikely to derive an NC algorithm from their sequential algorithm because they make use of the weighted matroid intersection algorithm by Gabow [62] which is highly sequential in nature.

### 5.6 Conclusions

In this chapter, firstly, we presented the first NC approximation algorithms for both the optimal 2-edge connectivity augmentation problem and the optimal biconnectivity augmentation problem. The solutions delivered by the two NC algorithms are \((1 + \ln n)(1 + \epsilon)\) times optimum and \((1 + \ln n_0)(1 + \epsilon) \log n_0 \) times optimum respectively when \( G(V, E_0) \) is connected, where \( n_0 \) and \( n_0 \) are the number of 2ECCs and 2VCCs of \( G(V, E_0) \) respectively, and \( \epsilon \) is a small constant with \( 0 < \epsilon < 1 \). Then, we gave an NC approximation algorithm for the optimal \( k \)-edge connectivity augmentation problem with fixed \( k \) (> 2) when \( G \) is \((k-1)\)-edge connected, the solution delivered is within \( \beta t \) times optimum where \( \beta = (1 + \ln n)(1 + \epsilon), 1 \leq t \leq \lfloor \log_2^\frac{1}{n} \rfloor \), and \( \epsilon \) is a constant with \( 0 < \epsilon < 1 \). If \( G \) is \( \lambda \)-edge connected, and \( 1 \leq \lambda < k - 1 \), our NC algorithm delivers an approximation solution within \( \alpha \beta \) times optimum, where \( \beta = (1 + \ln n)(1 + \epsilon) \), \( \epsilon \) is a constant with \( 0 < \epsilon < 1 \), and \( \alpha \) is defined as follows: (i) if both \( k \) and \( \lambda \) are odd (even), then \( \alpha = \frac{k-\lambda}{2}(t+1) \); (ii) if \( k \) is even and \( \lambda \) is odd, then \( \alpha = t^\left[\frac{k-1}{2}\right] + \left[\frac{k-1}{2}\right] \); (iii) otherwise, \( \alpha = t^\left[\frac{k-1}{2}\right] + \left[\frac{k-1}{2}\right] \) where \( 1 \leq t \leq \lfloor \log_2^\frac{1}{n} \rfloor \).
Chapter 6

Finding $k$ Most Vital Edges in Minimum Spanning Trees

6.1 Introduction

Let $G(V,E)$ be an undirected, weighted, connected simple graph with vertex set $V$ and edge set $E$. Associated with each edge $e \in E$, there is a real valued weight $w(e)$. A minimum spanning tree (MST) of $G$ is a spanning tree with minimum total weight. For the sake of convenience, denote by $MST(G)$ the MST of $G$, and $w(MST(G))$ the total weight of $MST(G)$. The problem of finding an MST of $G$ has been well studied in the past two decades [2, 63]. The best sequential algorithm for the MST problem needs $O(m \log \beta(m,n))$ time [63], where $m = |E|$, $n = |V|$ and $\beta(m,n) = \min \{i \mid \log^i n \leq m/n\}$. In particular, when $m \geq n \log^{i_0} n$ for some constant $i_0$, $\beta(m,n)$ is a constant. The best parallel algorithms for the MST problem require $O(\log n \log \log n)$ time using $O(m+n)$ processors on an EREW PRAM [26], and $O(\log^2 n)$ time using $O(n^2/\log^2 n)$ processors on a CREW PRAM [22] for sparse and dense graphs respectively.

One closely related problem is the $k$ most vital edge problem which can be formally defined as follows. Let $S^* \subseteq E$ be a set of $k$ edges, the $k$ most vital edge problem with respect to an MST of $G$ (the $k$ MVE MST problem for short) is to find such a set $S^*$ that $w(MST(G(V,E - S^*)))$ is maximized. This problem has many practical applications including robust network design and communication protocol design in distributed computing [58, 83, 88, 93, 126]. Obviously, $k \leq \lambda \leq [m/n]$, where $\lambda$ is the edge connectivity of $G$. Otherwise, after deleting all edges in a minimum edge cut of $G$ with any other $k - \lambda$ edges, the remaining graph is disconnected, and there is no MST existing in this remaining graph. Therefore, in this chapter we assume that $G$ is $(k+1)$-edge connected at least. Without loss of generality, we also assume the weights associated with the edges in $G$ are distinct. As a result, the MST of $G(V,E - S)$ is unique for any $S \subseteq E$.

When $k = 1$, the problem becomes finding the single most vital edge problem, which has been extensively studied in the literature [83, 88, 93, 157]. Hsu et al. [83] first considered this problem, and gave $O(n^2)$ time and $O(m \log m)$ time sequential algorithms for dense and sparse graphs respectively. Iwano et al. [93] improved Hsu et al’s results by giving an $O(t_{MST} + \min \{m \alpha(m,n), m + n \log n\})$ time algorithm
where $t_{MST}$ is the time used to find an MST of $G$. Hsu et al. [88] later proposed two parallel algorithms for this problem on an EREW PRAM. One of their parallel algorithms requires $O(n^{1+x})$ time and $O(n^{1-x})$ processors with $0 < x < 1$. The other one requires $O(m \log (m/N)/(N + n \alpha(m,n) \log (m/n)))$ time and $O(N)$ processors where $N \leq (m \log m)/(n \alpha(m,n) \log (m/n))$. All their algorithms need $\Omega(\log^2 n)$ time clearly. Recently Shen [157] presented another parallel algorithm for this problem. His algorithm requires either $O(\log n)$ time and $O(m \log \log \log n/\log n)$ processors on a PRIORITY CRCW PRAM, or $O(\log n \log \log n)$ time and $O(m + n^2/(\log n \log \log n))$ processors on a CREW PRAM.

For the case $k > 1$, not much work has been done. The only results are due to Lin et al. [126], Frederickson et al. [58] and Shen [158]. Lin et al. have shown that a generalized version of this problem is NP-complete, where each edge in $E$ is assigned a removal cost and the total removal cost $B$ is bounded. But their proof does not imply the NP-completeness of our problem which is a special case of their general version. Frederickson et al. [58] recently proved that the $k$ MVE MST problem is NP-complete, and presented an approximation algorithm for it. Their approximation algorithm requires $O(\min \{km \log n + k^2 n \log n, km \log^2 n\})$ time, and the solution delivered by their algorithm is $\Omega(1/\log k)$ times optimal. Shen [158] explored this problem by giving an exact algorithm and a randomized, approximation algorithm. His exact algorithm needs $O(n^k m \log \beta(m,n))$ time when $k$ is fixed. Note that, if $k$ is fixed, there always exists a polynomial algorithm for the $k$ MVE MST problem which generates exact solutions.

In this chapter, using the notion of the sparse, weighted $k$-edge certificate (defined later), we present a better exact algorithm for the $k$ MVE MST problem. When $k \geq 4$ is fixed, we suggest efficient sequential and parallel algorithms for this problem. Our sequential algorithm runs in $O(n^{k+1})$ time which improves by an $O(m \log \beta(m,n)/n)$ factor in the time bound over Shen’s algorithm [158]. Our parallel algorithm runs in $O(\log n)$ time using $O(n^{k+1})$ processors on an EREW PRAM provided both the MST of $G$ and the sparse, weighted $(k+1)$-edge certificate of $G$ are given. In particular, for $k = 1$, we develop an NC algorithm on the EREW PRAM which requires $O(\log n \log \log n)$ time and $O(m + n^2/(\log n \log \log n))$ processors. This algorithm outperforms all previous results on this model. We also suggest another NC algorithm on the CREW PRAM which has the same time complexity as Shen’s algorithm [157] on the same model, but we use $O(m)$ processors rather than $O(m + n^2/(\log n \log \log n))$ processors by his algorithm. For $k = 2, 3$, we develop $O(n^2 \alpha(3n,n))$ and $O(n^3 \alpha(4n,n))$ time sequential algorithms respectively. By parallelizing these two algorithms, we obtain two NC algorithms for them which require $O(\log n \log \log n)$ time using $O(m + n^2/\log n)$ processors and $O(\log n \log \log n)$ time using $O(n^3/\log n)$ processors respectively, the NC algorithms run in a CREW PRAM. The parallel computational models involved in this chapter are the following: the EREW PRAM, the CREW PRAM, and the PRIORITY CRCW PRAM.

The rest of this chapter is organized as follows. In Section 6.2 we introduce the notion of sparse, weighted $k$-edge certificates, and show how to use it to solve the $k$ MVE MST problem on $G$. In Section 6.3 we present two NC algorithms for the single
most vital edge problem which run in the EREW PRAM and the CREW PRAM respectively. In Section 6.4 we devise sequential and parallel algorithms for cases $k = 2, 3$. In Section 6.5, we develop efficient sequential and parallel algorithms for the $k$ MVE MST problem with fixed $k$. A conclusion is given in Section 6.6.

6.2 Finding $k$ Most Vital Edges

Without loss of generality, we assume that the weight assigned to each edge in $G$ is distinct and hence the MST or the minimum spanning forest (MSF for short) of graph $G(V,E) - S$ is unique for every $S \subseteq E$.

A naive approach to attacking the $k$ MVE MST problem proceeds as follows. First enumerate all different $S$ of $k$ edges from the set $E$, and compute the weight of the MST of $G(V,E-S)$ for each $S$. Then find a $S_0$ of $k$ edges such that $w(MST(G(V,E-S_0))) \geq w(MST(G(V,E-S)))$ for all other $S \neq S_0$. Thus, $S_0$ is the solution of the problem which contains the $k$ most vital edges. There are $\binom{m}{k}$ different subsets of $k$ edges in $E$. So the time used for the $k$ MVE MST problem on $G$ is

$$T_{naive} = \binom{m}{k} t_{MST(G')}$$

(6.1)

where $G' = G(V,E-S)$ with $|S| = k$, and $t_{MST(G')}$ is the worst-case time bound to compute the MST of $G'$.

When $k$ is fixed, $T_{naive} = \binom{m}{k} t_{MST(G')} = O(m^{k+1} \log \beta(m,n))$ if the well known fastest sequential algorithm is employed [63].

In the following we show that there exists a better algorithm for this problem. The idea behind our algorithm is that we first extend the notion of sparse $k$-edge certificates of an unweighted, undirected graph [135, 136] to a weighted, undirected graph $G$, and define the sparse, weighted $k$-edge certificate of $G$. We then show that the $k$ MVE MST problem on $G$ is exactly equivalent to the $k$ MVE MST problem on the sparse, weighted $(k+1)$-edge certificate $U_{k+1}$ of $G$. As a result, instead of using $G$, we will use $U_{k+1}$ to find the $k$ most vital edges in $G$. Thus, we reduce the size of the searching space from $\binom{m}{k}$ to $\binom{(k+1)(n-1)}{k}$. Particularly when $k$ is fixed, this improvement on the algorithm’s performance is significant.

Now we define the sparse, weighted $k$-edge certificate of $G$ by extending the sparse $k$-edge certificate on an unweighted, undirected graph in Section 2.2 of Chapter 2.

**Definition 6.1** Let $G$ be a weighted, undirected graph, $T_1$ be an MSF of $G$, and $T_i$ be an MSF of $G_i = G - \bigcup_{j=1}^{i-1} T_j$ for $i > 1$. Denote by $U_i = \bigcup_{j=1}^{i} T_j$, the union of the MSFs $T_1, T_2, \ldots, T_i$. The graph $U_k$ is called the sparse, weighted $k$-edge certificate of $G$.

**Lemma 6.1** The graph $U_{k+1}$ defined above is $(k+1)$-edge connected if and only if $G$ is at least $(k+1)$-edge connected.

The proof of Lemma 6.1 is easy and omitted. Actually it is a corollary of Lemma 2.1. Recall that in the beginning of this chapter we already assumed that $G$ is $(k+1)$-
edge connected at least. Therefore, from now on we assume that \( U_{k+1} \) is \((k+1)\)-edge connected.

Having defined the sparse, weighted \((k+1)\)-edge certificate \( U_{k+1} \) of \( G \), we show the following lemma which is the key to developing our algorithms later.

**Lemma 6.2** If \( e \in E - S \) is not an edge in \( U_{k+1} \), then \( e \) does not belong to the MST of the graph \( G(V, E - S) \) for any \( S \subseteq E \), where \(|S| \leq k\).

**Proof.** Let \( e = (u, v) \in E \) but \( e \notin E(U_{k+1}) \) where \( E(U_{k+1}) \) is the edge set of graph \( U_{k+1} \). Because \( u \) and \( v \) are connected by \( e = (u, v) \), \( u \) and \( v \) must be in the same CC in \( G \), and in \( G_i = G - \bigcup_{j=1}^{i-1} T_j \) for all \( i \), \( 1 \leq i \leq k + 1 \). Therefore, there is a unique path \( P_i \) between \( u \) and \( v \) in each \( T_i \), \( 1 \leq i \leq k + 1 \). Since \( e = (u, v) \) was not chosen in any \( T_i \) \((1 \leq i \leq k + 1)\), \( w(e) \) must be larger than the weight of any edge in \( P_i \), i.e., \( w(e) > w(e') \), where \( e' \in \bigcup_{i=1}^{k+1} E(P_i) \) and \( E(P_i) \) denotes the edge set of \( P_i \).

Now we prove that \( e = (u, v) \) cannot be contained in the MST of \( G(V, E - S) \) for any \( S \subseteq E \), \(|S| \leq k \). For the purpose of contradiction, suppose \( e = (u, v) \) is contained in such an MST. Then, deleting \( e \) from this MST will induce a partition of \( V \), \( V = V_S \cup (V - V_S) \), where \( V_S \) and \( V - V_S \) are the vertex sets of subtrees containing \( u \) and \( v \) respectively. As we argued above, the edges between \( V_S \) and \( V - V_S \) form a cut which contains at least one edge from each \( P_i \), \( 1 \leq i \leq k + 1 \). Because \(|S| \leq k \), there must be an edge \( e^* \in E(P_i) - S \subseteq E(U_{k+1}) - S \subseteq E - S \), such that \( w(e) > w(e^*) \). Replacing \( e \) with \( e^* \) in the MST will result in a spanning tree in \( G(V, E - S) \) which has less weight than that of the MST. A contradiction. \( \Box \)

**Remark.** Lemma 6.2 is equivalent to saying that for any \( S \), \( S \subseteq E \) and \(|S| \leq k \), the MST of \( G(V, E - S) \) is entirely contained in \( U_{k+1} \).

Lemma 6.2 implies an algorithm for our problem. That is, instead of selecting a subset of \( k \) edges from the set \( E \) of \( m \) edges, we only need to select a subset of \( k \) edges from the set \( E(U_{k+1}) \) of at most \((k+1)(n-1)\) edges. Therefore we have

**Theorem 6.1** The \( k \)-MVE MST problem on a weighted, connected, undirected simple graph \( G = (V, E) \) can be solved in time \( O\left(\binom{kn}{k}\frac{kn \log \beta(kn, n)}{\log n}\right) \) for any \( k \geq 1 \).

**Proof.** The discussion is similar to that used in the naive algorithm. The only difference is that we use the sparse, weighted \((k+1)\)-edge certificate \( U_{k+1} \) of \( G \) instead of \( G \) itself. Let \( T_{\text{certi}} \) be the time complexity for solving the \( k \)-MVE MST problem on \( G \). Then \( T_{\text{certi}} = \binom{(k+1)(n-1)}{k} t_{MST}(G') \), where \( G' \) is a sparse, weighted, undirected graph induced by deleting \( k \) edges from \( U_{k+1} \), and \( t_{MST}(G') \) is the worst-case time used to find the MST of \( G' \). It is well known that the best sequential algorithm for the MST problem on \( G' \) requires \( t_{MST}(G') = O(kn \log \beta(kn, n)) \) time, the theorem follows. \( \Box \)

In particular, when \( k \) is fixed, the time complexity for the \( k \)-MVE MST problem on \( G \) is:

\[
T_{\text{certi}} = \binom{(k+1)(n-1)}{k} t_{MST}(G') = O(n^{k+1} \log \beta(kn, n)). \tag{6.2}
\]
It must be mentioned that Shen [158] also suggested an exact algorithm for the k MVE MST problem on G. His algorithm needs $O(n^k m \log \beta(m, n))$ time when k is fixed, which is obviously inferior to our result above. We later will show how to improve the time bound in Equation 6.2 further to $O(n^{k+1})$ by using the technique in Section 6.5, which improves by an $O(m \log \beta(m, n)/n)$ factor over the time bound of Shen’s algorithm.

Now let $T$ be the MST of $G$ and $e = (u, v)$ be a tree edge in $T$, the ith minimum weighted replacement edge $r_i(e)$ of $e$ is defined as follows. Delete $e$ from $T$. As a result, the vertex set $V$ is divided into two subsets $W$ and $V - W$ which are the vertex sets of subtrees containing $u$ and $v$ respectively. Let $Q = (W \times (V - W)) \cap E - \{e\}$, then $r_i(e)$ is the ith minimum weighted edge in $Q$. Obviously $w(r_i(e)) < w(r_j(e))$ if $1 \leq i < j \leq |Q|$.

**Lemma 6.3** Let $G(V, E)$ be an undirected, weighted simple graph, $V_1$ and $V_2$ be an arbitrary partition of the vertex set $V$, $V_1 \cup V_2 = V$ and $V_i \neq \emptyset$, $i = 1, 2$. Further let $Q = (V_1 \times V_2) \cap E$. Then the minimum weighted edge in $Q$ must be in the MST/MSF of $G$.

**Proof.** Let $T$ be the MST/MSF of $G$ and $e = (u, v)$ be the minimum weighted edge in $Q$. Assume that $e$ is not in $T$, then there must exist a unique path $P_{u,v}$ between $u$ and $v$ in $T$. Otherwise, the edge $e$ should be chosen as an edge of $T$. It is easy to see that $P_{u,v}$ contains at least an edge $e' \in Q$. Otherwise, there is no any edges connecting the vertices between $V_1$ and $V_2$. This obviously contradicts that $e$ connects $u$ and $v$ which are in $V_1$ and $V_2$ respectively. Let $e'$ be defined as above, by our initial assumption we know that $w(e) < w(e')$. Now we replace the tree edge $e'$ by the non-tree edge $e$. As a result, we obtain a new tree which has a smaller weight than that of $T$. This contradicts that $T$ is the MST/MSF of $G$. So, $e$ must be in $T$. \(\Box\)

**Lemma 6.4** Let $E(T) = \{e_1, e_2, \ldots, e_{n-1}\}$ be the edge set of $T$, and $r_j(e_i), e_i \in E(T)$, be defined as above, then $r_j(e_i) \in U_{k+1}$ for all $i$ and $j$, $1 \leq i \leq n - 1$ and $1 \leq j \leq k$.

**Proof.** For any edge $e \in E(T)$ and $e = (u, v)$, if we can show that $r_l(e) \in U_{l+1}$ for all $l$, $1 \leq l \leq k$, the lemma follows. In the rest part we will dedicate to prove that $r_l(e) \in U_{l+1}$. Assume that $r_j(e)$ is the first edge which is not in $U_{j+1}$, i.e., $r_1(e) \in U_2$, $r_2(e) \in U_3$, $\ldots$, $r_{j-1}(e) \in U_j$ but $r_j(e) \notin U_{j+1}$. Now we consider the graph $G' = G - U_j$. Obviously the edge $r_j(e)$ has been deleted from $G'$ for all $i$, assume that $r_0(e) = e$, $0 \leq i \leq j - 1$. Let $T_{j+1}$ be the MST/MSF of $G'$ by Definition 6.1, and $T_v$ and $\mathcal{T}_u$ be the subtrees containing $v$ and $u$ respectively after deletion of $e$ from $T_1$ (i.e., $T$). Let $V_1$ and $V_2$ be the vertex set of $T_v$ and $\mathcal{T}_u$. It is easy to see that $V_i \neq \emptyset$, $i = 1, 2$ and $V_1 \cup V_2 = V$. Let $Q$ be an edge set of $G'$ in which the endpoints of the edges are in $V_1$ and $V_2$ respectively. Then $r_j(e)$ is the minimum weighted edge in $Q$ because the edges $e, r_1(e), r_2(e), \ldots, r_{j-1}(e)$ have been deleted, and $r_j(e) \notin U_j$. By Lemma 6.3, $r_j(e)$ must be in $U_{j+1}$. Therefore, it must be in $U_{j+1} = U_j \cup T_{j+1}$. \(\Box\)

In the following we show how to find the $r_j(e_i)$ for all $i$ and $j$, $1 \leq i \leq n - 1$ and $1 \leq j \leq k$. 

\(\Box\)
Lemma 6.5 Let $T$ be the MST of $G$, and $E(T) = \{e_1, e_2, \ldots, e_{n-1}\}$ be the edge set of $T$. The calculation of $r_j(e_i)$ for all $i$ and $j$ can be done in $O(n^2)$ time, where $1 \leq i \leq n - 1$, $1 \leq j \leq k$ with fixed $k$.

Proof. By Lemma 6.4, we can use $U_{k+1}$ instead of $G$ to compute all $r_j(e_i)$, $1 \leq i \leq n - 1$, $1 \leq j \leq k$ with fixed $k$. The rest is to present the detailed implementation. For an edge $e = (u, v) \in E(T)$, the computation of all $r_j(e)$, $1 \leq j \leq k$ can be done as follows: delete the edge $e$ from $T$, as the discussion above, $T$ becomes two subtrees containing $u$ and $v$ respectively. Label the vertices in each such subtree by a unique identification. This costs $O(n)$ time. Let $W$ and $V - W$ be the vertex sets containing $u$ and $v$ respectively. Then we choose the $j$th minimum weighted edge from $Q' = (W \times (V - W)) \cap U_{k+1} - \{e\}$ which is exactly $r_j(e)$ by Lemma 6.4. This can be done in time $O(n)$ because $|Q'| \leq |U_{k+1}| < (k + 1)n$ with fixed $k$. Therefore it costs $O(k(k + 1)n) = O(n)$ time to compute all $r_j(e)$, $1 \leq j \leq k$. The computation of all $r_j(e_i)$, $1 \leq i \leq n - 1$ and $1 \leq j \leq k$, can be done in time $O(n^2)$. □

6.3 NC Algorithms for Finding the Single Most Vital Edge

In this section we present NC algorithms for the special case $k = 1$ on both the EREW PRAM and the CREW PRAM.

Let $T_1$ be the MST of $G$, $E(T_1)$ be the edge set of $T_1$. Define the replacement edge $r(e)$ of an edge $e \in E(T_1)$ as $r(e) = r_1(e)$ if $Q \neq \emptyset$. Otherwise, there does not exist a replacement edge for $e$, and $e$ is a bridge of $G$. In this case, we know that the remaining graph is disconnected after deleting $e$ from $G$, therefore no MST exists in this remaining graph. Thus, the edge $e$ is the single most vital edge. For the discussion convenience later, we set $w(r(e)) := +\infty$ if $e$ is a bridge of $G$.

Now we consider the single most vital edge problem on $G$. Lin et al. [126] observed that the single most vital edge must be in $T_1$. Iwano et al. [93] further showed that an edge $e^* \in E(T_1)$ is the single most vital edge if and only if it satisfies the following equation.

$$w(e^*) = \max \{w(r(e)) - w(e) \mid e \in E(T_1)\}$$

(6.3)

Equation 6.3 implies that the edge $r(e)$ is an edge of the MST in the remaining graph after deletion of $e$ from $G$. Obviously $e^*$ can be obtained easily by computing $r(e)$ for all $e \in E(T_1)$ in parallel. Thus, an NC algorithm for this problem depends on how to compute $r(e)$ for every $e \in E(T_1)$ efficiently. Let $T_2$ be the MST (MSF) of the graph $G - T_1$, and $E(T_2)$ be the edge set of $T_2$. Before we continue, we reproduce the following lemma by Iwano et al. [93].

Lemma 6.6 [93] Let $T_1$ and $E(T_1)$ be defined as above, $new(e)$ be an edge weight function such that $new(e) = 0$ for $e \in E(T_1)$, and $new(e) = B - w(e)$ for $e \not\in E(T_1)$
where $B = \max\{w(e) \mid e \in E\} + \delta$ and $\delta$ is a positive constant. Let MaxST($G$) be a maximum spanning tree with respect to the above defined weight function $new(e)$, and $E(\text{MaxST}(G))$ be the edge set of MaxST($G$). For an edge $e \in E(T_1)$, there is a replacement edge $r(e) \in E(\text{MaxST}(G))$; that is, $T_1 - \{e\} \cup \{r(e)\}$ is an MST of $G - \{e\}$.

Lemma 6.6 suggests an algorithm for computing all $r(e)$ from the tree MaxST($G$). Let $w(\text{MaxST}(G))$ denote the weight of MaxST($G$) and $n' = |E(\text{MaxST}(G))|$. Then

$$w(\text{MaxST}(G)) = n'B - \sum_{i=1}^{n'} w(e_i), \quad \text{where} \quad e_i \in E(\text{MaxST}(G)).$$

(6.4)

Notice that, for any given undirected, weighted simple graph $G''$ with $n''$ vertices no matter whether it is connected or not, the number of edges $n'$ in all spanning trees (or spanning forests) of $G''$ is the same. This means that the value $n'B$ in Equation 6.4 is fixed when $G$ is given. By Equation 6.4, we know that, in order to make $w(\text{MaxST}(G))$ maximized, we only need to find a spanning tree (or a spanning forest) $T'$ in $G - T_1$ such that $\sum_{i=1}^{n'} w(e'_i)$ is minimized where $e'_i \in E(T')$. By the definition of the MST (MSF), $T'$ is the MST (MSF) of $G - T_1$, i.e., $T' = T_2$. Obviously MaxST($G$) = $T'$ = $T_2$. Shen [157] noticed this and presented it explicitly by the following lemma. Note that the edges in $T_1$ are not included in MaxST($G$) because the new weights associated with the edges in $T_1$ are zeros, whereas all the new weights associated with the edges in MaxST($G$) are positive by Lemma 6.6.

**Lemma 6.7** [157] For any edge $e$ in $T_1$, $r(e)$ is an edge in the MST of the graph $G - T_1$.

Assume that $T_1$ is a rooted tree. Now we present an approach for computing $r(e)$ for each $e \in E(T_1)$. We do the pre-order traversal on $T_1$ and assign to each vertex $v$ the pre-order numbering $pre(v)$ and the number of descendants $nd(v)$ (including $v$ itself). Having done the above, we now consider how to compute $r(e)$ for every edge $e = (u, v) \in E(T_1)$. Suppose that $v$ is the parent of $u$ in $T_1$. By Lemma 6.7, for an edge $e \in E(T_1)$ we have the following formula.

$$w(r(e)) = \min\{w(e'') \mid e'' = (u', v') \in E(T_2), \quad pre(u) \leq pre(u') < pre(u) + nd(u),$$

and either $pre(v') < pre(v')$ or $pre(v') \geq pre(v) + nd(u)\}.\quad (6.5)$$

In the following we first present an NC algorithm for this problem on an EREW PRAM. In order to compute $r(e)$ for each $e \in E(T_1)$, we need a copy of $T_2$, and a copy of both $pre(v)$ and $nd(v)$ for every $v \in V$ because we use the EREW PRAM. So, there are a total of $n - 1$ copies of $T_2$, $pre(v)$ and $nd(v)$ needed. This can be implemented in $O(\log n)$ time using $O(n^2/\log n)$ processors on this model by the broadcasting technique. It can also be implemented in $O(\log n \log \log n)$ time using $O(n^2/(\log n \log \log n))$ processors on this model by Brent’s Theorem. Now for each edge $e \in E(T_1)$, there is a corresponding copy of $T_2$ as well as a copy of $pre(v)$ and $nd(v)$ for each $v \in V$. By Equation 6.5 above, $r(e)$ can be obtained in $O(\log n)$ time using $O(n/\log n)$ processors.
on an EREW PRAM because of $|E(T_2)| \leq n - 1$. Note that $r(e)$ can also be obtained in $O(\log n \log \log n)$ time using $O(n/(\log n \log \log n))$ processors on the same model by Brent’s Theorem. Thus, we have the following theorem.

**Theorem 6.2** Given a weighted, connected, undirected simple graph $G(V,E)$, the single most vital edge can be found in $O(\log n \log \log n)$ time using $O(m + n^2/(\log n \log \log n))$ processors on an EREW PRAM.

**Proof.** Finding the trees $T_1$ and $T_2$ require $O(\log n \log \log n)$ time using $O(m)$ processors on an EREW PRAM respectively, assuming $m \geq n - 1$. It is well known that the assignment of pre-order numbering $\text{pre}(v)$ and the number of descendents $\text{rd}(v)$ of $v$ in a rooted tree can be done in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM by Euler traversal and tree contraction techniques [1]. Given $T_2$, $\text{pre}(v)$ and $\text{rd}(v)$ for each vertex $v$ in $T_1$, the computation of all replacement edges in $T_1$ can be done in $O(\log n \log \log n)$ time using a total of $O(n^2/(\log n \log \log n))$ processors on an EREW PRAM. Then, Equation 6.3 can be computed in $O(\log n)$ time with $O(n/\log n)$ processors. Therefore, the theorem follows. □

Next we consider how to compute $r(e)$ for each $e \in E(T_1)$ on a CREW PRAM. In this model we do not need to make $n - 1$ copies of $T_2$ initially. We proceed by the following lemma.

**Lemma 6.8** Let $T'$ be the MST of a weighted undirected graph $G'(V,E')$ and $E(T')$ be the edge set of $T'$, then all $r(e_i) (= v_1(e_i)), e_i \in E(T')$, can be figured out in time $O(\alpha(m',n)(m' + n))$ at most, or in time $O(\alpha(m',n) \log n)$ using $O((m' + n)/\log n)$ processors on a CREW PRAM, where $1 \leq i \leq n - 1$, $|V| = n$ and $|E'| = m'$.

**Proof.** From the algorithm for the sensitivity analysis of MSTs due to Dixon et al. [41], we know that one important part in this algorithm is to compute $r_1(e_i)$ (which was defined as $b(u_i,v_i)$ in that paper) for all $e_i = (u_i,v_i) \in E(T')$. The time upper bound of their algorithm is $O((m' + n)\alpha(m' + n))$. Therefore, the time used for computing all $r_1(e_i)$ is $O((m' + n)\alpha(m' + n))$ at most, $1 \leq i \leq n - 1$. Later Dixon et al. [42] gave a parallel version of the sensitivity analysis algorithm. Their parallel algorithm requires $O(\alpha(m',n) \log n)$ time and $O((m' + n)/\log n)$ processors on a CREW PRAM. Thus, the computation of all $r_1(e_i)$ can be done with in the same time and processor bounds on this model, $1 \leq i \leq n - 1$. □

Based on Lemma 6.8, we now state the following theorem.

**Theorem 6.3** Given a weighted, connected, undirected simple graph $G(V,E)$, the single most vital edge can be found in $O(\log n \log \log n)$ time using $O(m)$ processors on a CREW PRAM.

**Proof.** The construction of $U_2$ requires $O(\log n \log \log n)$ time and $O(m + n)$ processors on an EREW PRAM using the fastest MST algorithm on this model. $U_2$ also can be found in the same time and processor bounds on a CREW PRAM because the EREW
PRAM is more restricted, compared with the CREW PRAM. The computation of all \( r(e) \) can be done in \( O(\alpha(2n, n) \log n) \) time using \( O(n/ \log n) \) processors on a CREW PRAM by Lemma 6.8, where \( e \in E(T) \). Notice that \( \alpha(2n, n) \leq c \log \log n \), where \( c \) is a constant. Equation 6.3 can be computed in \( O(\log n) \) time using \( O(n/ \log n) \) processors on a CREW PRAM. Since \( G \) is connected and \( m > n - 1 \), the theorem follows. \( \Box \)

The time complexity of our algorithm on the EREW PRAM is a substantial improvement on that in Hsu et al's algorithm which needs \( \Omega(\log^2 n) \) time on the same model. The cost to achieve our time bound is by employing more processors and incorporating currently the most efficient MST algorithm [26]. However, the number of processors in our algorithm is no more than \( O(n^2) \). With the same time-processor product, our algorithm is much faster. The time complexity of our algorithm in the CREW PRAM is the same as Shen's [157] on the model, but we use \( O(m) \) processors rather than \( O(m + n^2/(\log n \log \log n)) \) processors in his algorithm. Particularly, when we consider sparse graphs, our algorithm is much better than Shen's algorithm. For example, consider a sparse graph with \( n \) vertices and \( m = O(n) \) edges, our algorithm requires \( O(\log n \log \log n) \) time and \( O(n) \) processors on the model, but his algorithm needs \( O(n^2/(\log n \log \log n)) \) processors with the same time complexity.

### 6.4 Sequential and Parallel Algorithms for \( k = 2, 3 \)

#### 6.4.1 Finding the first two most vital edges

**Lemma 6.9** Let \( T \) be the MST of \( G \), and \( E(T) \) be the edge set of \( T \). Assume that \( S^* \) is the set of \( k \) edges whose deletion results in the maximum increase in the weight of the MST of \( G(V, E - S^*) \). Then \( E(T) \cap S^* \neq \emptyset \).

**Proof.** The proof of this lemma is easy. The approach adopted is similar to the one used in [83]. If \( E(T) \cap S^* = \emptyset \), then \( G(V, E - S^*) \) contains the tree \( T \). Thus the MST of \( G(V, E - S^*) \) is \( T \). However, replacing any edge \( e \in E(T) \) with \( r(e) \) will result in a spanning tree of larger weight. A contradiction. \( \Box \)

Since \( |S^*| = 2 \), by Lemma 6.9, we have either (i) \( |S^* \cap E(T)| = 1 \); or (ii) \( |S^* \cap E(T)| = 2 \). We now analyze these two cases separately. Assume that all \( r_j(e_i) \) have been calculated for all \( i \) and \( j \) with \( 1 \leq i \leq n - 1 \) and \( 1 \leq j \leq 2 \).

When \( |S^* \cap E(T)| = 1 \), then only one tree edge is one of the first two most vital edges. Let \( e^* \in E(T) \) be such a tree edge, then we have

**Lemma 6.10** Let \( S^* \) and \( T \) defined as above. If \( |S^* \cap E(T)| = 1 \), i.e., \( e^* \in S^* \cap E(T) \), then another most vital edge in \( S^* \) must be \( r_1(e^*) \).

**Proof.** Assume that \( r_1(e^*) \not\in S^* \), then the MST of \( G(V, E - S^*) \) is \( T'(V, E(T) - \{e^*\} \cup \{r_1(e^*)\}) \), and the weight of \( T' \) is \( w(MST) - w(e^*) + w(r_1(e^*)) \) by the definition of MSTs. Now consider a subgraph \( G' \) of \( G \) after deleting both \( e^* \) and \( r_1(e^*) \) from \( G \), then the weight of an MST of \( G' \) is \( w(MST) - w(e^*) - w(r_1(e^*)) + w(r_2(e^*)) \)
> w(MST) − w(e*) + w(r_1(e*)) because w(r_2(e*)) > w(r_1(e*)) by the definition in Section 6.2, which contradicts our assumption that r_1(e*) does not belong to S*. □

Now we discuss the case of |S* ∩ E(T)| = 2, which means both of the first two most vital edges are the tree edges. In order to find these two edges, the approach adopted is as follows: first, delete arbitrary two edges e_i and e_j from T, i ≠ j, then recompute the MST of the remaining graph G(V, E \{e_i, e_j\}) (actually we use U_3 \{e_i, e_j\} instead of G(V, E \{e_i, e_j\}) by Lemma 6.2). Let W_{e_i,e_j} be the weight of the MST of the remaining graph after the deletion of edges e_i and e_j from G. As a result, we choose one combination from all possible combinations of two edges among the n−1 tree edges such that the remaining MST has the maximum weight, which is the set of the first two most vital edges. Here the key is how to calculate the MST of the remaining graph quickly after deleting the two tree edges from G. In the following we give an approach for this propose.

By the above discussion and Lemmas 6.4, 6.5, 6.8, 6.9, and 6.10, we have the following the detailed algorithm for the problem. Let the variable W_{max} contain the weight of the final MST in the remaining graph, and e_1* and e_2* be the first two most vital edges.

Procedure Find_Two_Vital_Edges(G,V,E)
1. find the MST T and the sparse, weighted 3-edge certificate U_3 of G;
2. W_{max} := w(MST);
3. for each edge e_i ∈ E(T) do
   compute the edge r_j(e_i), j = 1, 2;
   /* this can be done using the subgraph U_3 of G. */
   endfor;
4. for each edge e_i ∈ E(T) do
   if w(MST) − w(e_i) − w(r_1(e_i)) + w(r_2(e_i)) > W_{max} then
      W_{max} := w(MST) − w(e_i) − w(r_1(e_i)) + (w(r_2(e_i)));
      e_1* := e_i; e_2* := r_1(e_i);
   endif;
endfor;
/* Steps 5 and 6 correspond to Case (ii) */
5. for each edge e_i; 1 ≤ i ≤ n − 1 do
   5.1. form a tree T_{e_i} by copying T;
   5.2. update the tree T_{e_i} by deleting the edge e_i from it;
   5.3. W_{MST}(e_i) := w(MST) − w(e_i) + w(r_1(e_i));
   /* W_{MST}(e_i) is used to store the weight of T_{e_i} */
   5.4. for each edge e_j; 1 ≤ j ≤ n − 1 and j ≠ i do
      find the first minimum weighted replacement edge e_j'
      for e_j from T_{e_i} by Dixon et al’s algorithm [41];
   endfor;
   5.5. for each edge e_j; 1 ≤ j ≤ n − 1 and j ≠ i do
      if W_{MST}(e_i) − w(e_j) − w(e_j') > W_{MST}(e_i) then
        /* the edge e_j has invalid property */
        endfor;
endfor;
/* Step 6 corresponds to Case (i) */
\[
\begin{align*}
W_{MST}(e_i) & := W_{MST}(e_i) - w(e_j) + w(e'_j); \\
\text{edge}(e_i) & := e_j; \\
/ \ast \text{ edge}(e_i) \text{ is possibly another vital edge }/^{\ast}
\end{align*}
\]

```c
e = 0;
```

```c
e = 0;
```

```c
for each edge \( e_i : 1 \leq i \leq n - 1 \) do
```

```c
if \( W_{MST}(e_i) > W_{max} \) then
```

```c
W_{max} := W_{MST}(e_i);
```

```c
e_1 := e_i; e_2 := \text{edge}(e_i);
```

```c
endfor;
```

```c
endfor;
```

The correctness of the proposed algorithm is justified by the above lemmas. Now we analyze the time complexity of this algorithm.

**Theorem 6.4** Given a weighted undirected graph \( G(V, E) \), the first two most vital edge problem w.r.t. MSTs can be solved in time \( O(n^2 \alpha(3n, n)) \).

**Proof.** By Algorithm \textit{Find\_Two\_Vital\_Edges}, Step 1 can be done in \( O(m \log \beta(m, n)) = O(n^2) \) time; Step 2 is a initial assignment which can be done in \( O(1) \) time; Step 3 can be done in time \( O(n^2) \) by Lemma 6.4. Step 4 needs \( O(n) \) time. Step 5 is the dominant step which is analyzed as follows. Step 5.1 requires \( O(n) \) time; Step 5.2 costs \( O(n) \) time by replacing \( e_i \) with \( r(e_i) \); Step 5.3 needs constant time; Step 5.4 requires \( O(n \alpha(3n, n)) \) time by Lemmas 6.2, 6.4 and 6.8; Step 5.5 costs \( O(n) \) time. Therefore Step 5 can be finished in time \( O(n^2 \alpha(3n, n)) \). Obviously Step 6 needs \( O(n) \) time. The algorithm thus requires \( O(n^2 \alpha(3n, n)) \) time. \( \square \)

**Theorem 6.5** Given a weighted undirected graph \( G(V, E) \), the first two most vital edge problem w.r.t. MSTs can be solved in \( O(\log n \log \log n) \) time using \( O(m + n^2 / \log n) \) processors on a CREW PRAM.

**Proof.** The proposed algorithm \textit{Find\_Two\_Vital\_Edges} can be parallelized easily. In the following we analyze the computational complexities of its parallelization. Step 1 requires \( O(\log n \log \log n) \) time and \( O(m + n) \) processors on an EREW PRAM \cite{26}. Step 2 costs \( O(1) \) time. Step 3 needs \( O(\log n) \) time \( O(n^2 / \log n) \) processors. Its implementation details are as follows. For every edge \( e \in E(T) \), make a copy of \( T \). Let this copy be \( T_e \) rooted at \( r \), delete \( e \) from \( T_e \), using the algorithm in \cite{1} to label each subtree. This can be done in \( O(\log n) \) time using \( O(n / \log n) \) processors on an EREW PRAM. Label the endpoints of each edge in \( U_3 \) by its subtree identification which requires \( O(1) \) time and \( O(|U_3|) = O(n) \) processors on a CREW PRAM. Find the minimum weighted edge and the second minimum weighted edge from the set consisting of edges in \( U_3 \) whose endpoints are labeled by different subtree identifications.
It is obvious that this step can be done in \(O(\log n)\) time using \(O(n/\log n)\) processors on an EREW PRAM. So, finding all \(r_j(e)\) requires \(O(\log n)\) time using \(O(n^2/\log n)\) processors on a CREW PRAM for all \(e \in E(T)\) and \(j = 1, 2\). Step 4 needs \(O(1)\) time and \(O(n)\) processors at most. The analysis of Step 5 is as follows. Step 5.1 requires \(O(1)\) time and \(O(n)\) processors; Step 5.2 requires \(O(\log n)\) time using \(O(n)\) processors on an EREW PRAM; Step 5.3 needs \(O(1)\) time and \(O(1)\) processors; Step 5.4 requires \(O(\alpha(3n, n) \log n)\) time and \(O(n/\log n)\) processors on a CREW PRAM by Lemma 6.8; Step 5.5 and Step 6 needs \(O(1)\) time and \(O(n)\) processors. Therefore, the proposed algorithm requires \(O(\alpha(3n, n) \log n + n \log n \log \log n) = O(\log n \log \log n)\) time using \(O(n + n^2 / \log n)\) processors on a CREW PRAM because \(\log \log n > \alpha(3n, n)\) when \(n\) is enough large. □

### 6.4.2 Finding the first three most vital edges

The idea of the algorithm for finding the first three most vital edges is basically the same as that for finding the first two most vital edges. Recall that \(T\) is the MST of \(G, \text{and } E(T) = \{e_1, e_2, \ldots, e_{n-1}\}\) is the edge set of \(T\). Let \(S^*\) be the set of the first three most vital edges. Then, by Lemma 6.9, one of the following cases must be held: (i) \(|S^* \cap E(T)| = 1\); (ii) \(|S^* \cap E(T)| = 2\); or (iii) \(|S^* \cap E(T)| = 3\). We now analyze the three cases separately. Assume that \(r_j(e_i)\) has been calculated for all \(i\) and \(j\) with \(1 \leq i \leq n - 1\) and \(1 \leq j \leq 3\).

When \(|S^* \cap E(T)| = 1\), only a tree edge is one of the first three most vital edges. Let \(e^* \in E(T)\) be such a tree edge. Then we have

**Lemma 6.11** Let \(S^*\) be the set of the first three most vital edges and \(T\) be defined as above. If \(|S^* \cap E(T)| = 1\), i.e., \(e^* \in S^* \cap E(T)\), then the other two most vital edges in \(S^*\) must be \(r_1(e^*)\) and \(r_2(e^*)\).

**Proof.** The proof is similar to that used in Lemma 6.10, omitted. □

Next we discuss the case of \(|S^* \cap E(T)| = 2\), which means that the two of the first three most vital edges are the tree edges.

**Lemma 6.12** Let \(S^*\) be the set of the first three most vital edges and \(T\) be defined as above. Assume that \(S^* \cap E(T) = \{e_1^*, e_2^*, e_3^*\}\). Then the third most vital edge in \(S^*\) is the minimum weighted edge in either \(Q_1\), \(Q_2\), or \(Q_3\), where \(Q_1\), \(Q_2\), and \(Q_3\) are the edge set which are defined as follows. After deleting \(e_1^*\) and \(e_2^*\) from \(T\), the vertex set \(V\) is divided into three subsets \(V_1, V_2, \text{ and } V_3\) which correspond to the three subtrees. Then \(Q_1 = (V_1 \times V_2) \cap E = (V_1 \times V_2) \cap U_4, Q_2 = (V_2 \times V_3) \cap E = (V_2 \times V_3) \cap U_4, \text{ and } Q_3 = (V_1 \times V_3) \cap E = (V_1 \times V_3) \cap U_4\).

**Proof.** Let \(e_i^{(1)}\) be the minimum weighted edge in \(Q_i, i = 1, 2, 3\). In the following we only deal with the case \(w(e_3^{(1)}) > \max\{w(e_1^{(1)}), w(e_2^{(1)})\}\). The other two cases \(w(e_2^{(1)}) > \max\{w(e_1^{(1)}), w(e_3^{(1)})\}\) and \(w(e_1^{(1)}) > \max\{w(e_2^{(1)}), w(e_3^{(1)})\}\) can be handled
similarly and are omitted. if \( w(e_3^{(1)}) > \max\{w(e_1^{(1)}), w(e_2^{(1)})\} \), then either \( e_1^{(1)} \) or \( e_2^{(1)} \) is the third most vital edge. Assume neither one is the third most vital edge, then the weight of the MST of the graph after deleting all edges in \( S^* \) from \( G \) is \( w(MST) - w(e_1^{(1)}) - w(e_2^{(1)}) + w(e_1^{(1)}) + w(e_2^{(1)}) \). Now consider an MST of the graph after deleting \( e_1^{*}, e_2^{*}, e_1^{(1)} \) from \( G \), the weight of the MST is \( w(MST) - w(e_1^{(1)}) - w(e_2^{(1)}) + w(r_1(e_1^{(1)})) + w(e_2^{(1)}) \) which is larger than that of the MST of \( G(V, E - S^*) \), where \( r_1(e_1^{(1)}) \) is the minimum weighted replacement edge of \( e_1^{(1)} \) of \( G(V, E - \{e_1^{*}, e_2^{*}\}) \). This contradicts the assumption of that \( S^* \) is the set of the first three most vital edges. Therefore the third most vital edge must be either \( e_1^{(1)} \) or \( e_2^{(1)} \) for this case. Similarly, for the other two cases, we can show that the third most vital edge is either \( e_1^{(1)} \) or \( e_3^{(1)} \). The lemma thus follows. □

Last we consider the case \( S^* \cap E(T) = \{e_1^{*}, e_2^{*}, e_3^{*}\} \). In this case the first three most vital edges are the tree edges. In the rest we present how to find the three edges. Assume that \( e_1^{*} \) and \( e_2^{*} \) have already been found. The question is to find \( e_3^{*} \). Let \( T_{e_1^{*}e_2^{*}} \) be the MST of \( G(V, E - \{e_1^{*}, e_2^{*}\}) \). Then \( e_3^{*} \) is a tree edge in \( T_{e_1^{*}e_2^{*}} \) and \( e_3^{*} \in E(T) \) whose deletion results in the maximum increase of the weight of the final MST. So, we can apply the algorithm of Dixon et al. [41] to find the 1st minimum weighted replacement edge for each edge in \( T_{e_1^{*}e_2^{*}} \). The algorithm is presented as follows. Let the variable \( W_{max} \) contain the weight of the MST in the remaining graph, and \( e_1^{*}, e_2^{*} \) and \( e_3^{*} \) be the first three most vital edges.

Procedure **Find_Three_Vital_Edges(G,V,E)**

1. find the MST \( T \) and the sparse, weighted 4-edge certificate \( U_4 \) of \( G \);
2. \( W_{max} := w(MST) \);
3. for each edge \( e_i \in E(T) \) do
   compute the edge \( r_j(e_i) \), \( j = 1, 2, 3 \);
   /* this can be done using the subgraph \( U_4 \) of \( G \). */
   endfor;
   /* Step 4 corresponds to Case (i) */
4. for each edge \( e_i \in E(T) \) do
   if \( w(MST) - w(e_i) - w(r_1(e_i)) - w(r_2(e_i)) + w(r_3(e_i)) > W_{max} \) then
     \( W_{max} := w(MST) - w(e_i) - w(r_1(e_i)) - w(r_2(e_i)) + (w(r_3(e_i)));
     e_1^{*} := e_i; e_2^{*} := r_1(e_i); e_3^{*} := r_2(e_i);
   endif;
   endfor;
   /* Step 5 corresponds to Case (ii) */
5. for each edge \( e_i \in E(T) \) do
   for each edge \( e_j \in E(T) \) do
     if \( e_i \neq e_j \) then
       5.1. delete \( e_i \) and \( e_j \) from \( T \), find the minimum
           weighted edge \( e_k^{(1)} \) from \( Q_k \), \( k = 1, 2, 3 \);
       5.2. re-build the MST \( T_{e_k^{(1)}e} \) for \( G(V, E - \{e_i, e_j\}) \);
       5.3. compute \( r_1(e_k^{(1)}) \) for \( e_k^{(1)} \) from graph \( U_4 - \{e_i, e_j\} \), \( k = 1, 2, 3 \);
       5.4. if \( w(e_3^{(1)}) > \max \{w(e_1^{(1)}), w(e_2^{(1)})\} \) then

/* this means that $e_3^{[1]}$ is not chosen as a tree edge */
if \( w(e_1^{[1]}) + w(r_1(e_2^{[1]})) > w(r_1(e_1^{[1]})) + w(e_1^{[1]}) \) then
    \( e_{\text{temp}} := e_2^{[1]} \); \( W_{\text{temp}} := w(e_1^{[1]}) + w(r_1(e_2^{[1]})) \)
else \( e_{\text{temp}} := e_1^{[1]} \); \( W_{\text{temp}} := w(r_1(e_1^{[1]})) + w(e_1^{[1]}); \)
endif;
else
    if \( w(e_2^{[1]}) > \max\{w(e_1^{[1]}), w(e_3^{[1]})) \) then
        /* this means that $e_2^{[1]}$ is not chosen as a tree edge */
        if \( w(e_1^{[1]}) + w(r_1(e_3^{[1]})) > w(r_1(e_1^{[1]})) + w(e_3^{[1]} \) then
            \( e_{\text{temp}} := e_3^{[1]} \); \( W_{\text{temp}} := w(e_1^{[1]}) + w(r_1(e_3^{[1]})) \)
        else \( e_{\text{temp}} := e_1^{[1]} \); \( W_{\text{temp}} := w(r_1(e_1^{[1]})) + w(e_3^{[1]}); \)
        endif;
    endif;
else
    /* this means that $e_1^{[1]}$ is not chosen as a tree edge */
    if \( w(e_2^{[1]}) + w(r_1(e_3^{[1]})) > w(r_1(e_2^{[1]})) + w(e_3^{[1]} \) then
        \( e_{\text{temp}} := e_3^{[1]} \); \( W_{\text{temp}} := w(e_2^{[1]}) + w(r_1(e_3^{[1]})) \)
    else \( e_{\text{temp}} := e_2^{[1]} \); \( W_{\text{temp}} := w(r_1(e_2^{[1]})) + w(e_3^{[1]}; \)
    endif;
endif;
5.5. if \( w(\text{MST}) - w(e_i) - w(e_j) - w(e_{\text{temp}}) + W_{\text{temp}} > W_{\text{max}} \) then
    \( W_{\text{max}} := w(\text{MST}) - w(e_i) - w(e_j) - w(e_{\text{temp}}) + W_{\text{temp}} \);
    \( e_1^* := e_i \); \( e_2^* := e_j \); \( e_3^* := e_{\text{temp}} \);
endif;
endfor;
/* Steps 6 and 7 correspond to Case (iii) */
6. for each edge \( e_i \in E(T) \) do
    for each edge \( e_j \in E(T) \) do
        6.1. compute the weight \( W_{T_{e_i,e_j}} \) of \( T_{e_i,e_j} \) if \( i \neq j \);
        6.2. find \( r_1(e_k^i) \) for all \( e_k^i \in E(T_{e_i,e_j}), 1 \leq k \leq n - 1 \);
    endfor;
endfor;
7. for each edge \( e_i \in E(T) \) do
    for each edge \( e_j \in E(T) \) do
        for each edge \( e_k^i \in E(T_{e_i,e_j}) \cap E(T) \) do
            if \( W_{T_{e_i,e_j}} - w(e_k^i) + w(r_1(e_k^i)) > W_{\text{max}} \) then
                \( W_{\text{max}} := W_{T_{e_i,e_j}} - w(e_k^i) + w(r_1(e_k^i)) \);
                \( e_1^* := e_i \); \( e_2^* := e_j \); \( e_3^* := e_k^i \);
            endif;
        endfor;
    endfor;
endfor;
endfor;
endfor.

The correctness of the proposed algorithm is justified by the above discussion. Now we analyze the time complexity of algorithm $\text{Find\_Three\_Vital\_Edges}$.

**Theorem 6.6** Given a weighted undirected graph $G(V, E)$, the first three most vital edge problem w.r.t. MSTs can be solved in time $O(n^3 \alpha(4n, n))$.

**Proof.** By Algorithm $\text{Find\_Three\_Vital\_Edges}$, Step 1 can be done in time $O(m \log \beta(m, n)) = O(n^2)$; Step 2 is an initial assignment which can be done in $O(1)$ time. Step 3 can be done in time $O(n^2)$ by Lemma 6.4. Step 4 needs $O(n)$ time. The analysis of Step 5 is as follows. Step 5.1 needs $O(n)$ time because $|U_4| < 4n$; Step 5.2 costs $O(n)$ time; Step 5.3 can be done in $O(n)$ time; Steps 5.4 and 5.5 take constant time. So, Step 5 takes $O(n^3)$ time. Step 6 is the dominant step which is analyzed as follows. Step 6.1 requires $O(n)$ time; Step 6.2 costs $O(n \alpha(4n, n))$ time by Lemma 6.8. Step 6 thus takes $O(n^3 \alpha(4n, n))$ time. Step 7 costs $O(n^3)$ time. The algorithm thus requires $O(n^3 \alpha(4n, n))$ time. \(\Box\)

**Theorem 6.7** Given a weighted undirected graph $G(V, E)$, the first three most vital edge problem w.r.t. MSTs can be solved in $O(\log n \log \log n)$ time using $O(n^3 / \log n)$ processors on a CREW PRAM.

**Proof.** The proposed algorithm $\text{Find\_Three\_Vital\_Edges}$ can be parallelized easily. In the following we analyze the computational complexities of its parallelization. Step 1 requires $O(\log n \log \log n)$ time and $O(m + n)$ processors on an EREW PRAM [26]. Step 2 costs $O(1)$ time. Step 3 needs $O(\log n)$ time $O(n^2 / \log n)$ processors. Its implementation details are as follows. For every edge $e \in E(T)$, make a copy of $T$. Let this copy be $T_e$ rooted at $r$, delete $e$ from $T_e$, using the algorithm in [1] to label each subtree. This can be done in $O(\log n)$ time using $O(n / \log n)$ processors on an EREW PRAM. Label the endpoints of each edge in $U_4$ by its subtree identification which requires $O(1)$ time and $O(|U_4|) = O(n)$ processors on a CREW PRAM. Find the 1st, 2nd and 3rd minimum weighted edges from the set consisting of edges in $U_4$ whose endpoints are labeled by different subtree identifications. It is obvious that this step can be done in $O(\log n)$ time using $O(n / \log n)$ processors on an EREW PRAM. So, finding all $r_j(e)$ requires $O(\log n)$ time using $O(n^2 / \log n)$ processors on a CREW PRAM for all $e \in E(T)$ and $j = 1, 2, 3$. Step 4 needs $O(1)$ time and $O(n)$ processors. The analysis of Step 5 is as follows. Step 5.1 requires $O(\log n)$ time and $O(n / \log n)$ processors whose implementation details are similar to that used in Step 3; Step 5.2 requires $O(\log n)$ time using $O(n / \log n)$ processors on an EREW PRAM; Step 5.3 requires $O(\log n)$ time using $O(n / \log n)$ processors on a CREW PRAM; Steps 5.4 and 5.5 need $O(1)$ time and $O(1)$ processors. Therefore Step 5 needs $O(\log n)$ and $O(n^3 / \log n)$ processors on a CREW PRAM. Step 6 is the dominant step. Step 6.1 requires $O(\log n)$ time and
\(O(n/\log n)\) processors on an EREW PRAM; Step 6.2 runs in \(O(\alpha(4n, n) \log n)\) time using \(O(n/\log n)\) processors on a CREW PRAM by Dixon et al's algorithm [42]. Step 6 thus requires \(O(\alpha(4n, n) \log n)\) time using \(O(n^3/\log n)\) processors on a CREW PRAM. Step 7 requires \(O(1)\) time and \(O(n^3)\) processors on a CREW PRAM, or \(O(\log n)\) time using \(O(n^3/\log n)\) processors on the same model by the famous Brent's Theorem. Therefore, the proposed algorithm requires \(O(\alpha(4n, n) \log n + \log n \log \log n)\) = \(O(\log n \log \log n)\) time using \(O(n^3/\log n)\) processors on a CREW PRAM because \(\log \log n > \alpha(4n, n)\) when \(n\) is enough large. \(\square\)

6.5 Sequential and Parallel Algorithms with Fixed \(k \geq 4\)

In this section we provide efficient sequential and parallel algorithms for the \(k\) MVE MST problem with fixed \(k \geq 4\).

By Theorem 6.1, we can easily suggest a parallel algorithm for the \(k\) MVE MST problem with arbitrary \(k\). The algorithm can be described as follows. First, find the sparse, weighted \((k+1)\)-edge certificate \(U_{k+1}\) of \(G\) which can be done in \(O((k \log n \log \log n)\) using \(O(m + n)\) processors on an EREW PRAM by applying the best parallel algorithm for the MST problem [26]. Then select a set \(S\) of \(k\) edges from \(E(U_{k+1})\) arbitrarily, and delete all edges of \(S\) from \(U_{k+1} \). Third, compute the MST of the remaining graph \(U_{k+1} - S\). This can be done in \(O(\log n \log \log n)\) time using \(O(kn)\) processors on an EREW PRAM. There are a total of \(\binom{(k+1)(n-1)}{k}\) different \(S\) sets, thus, the total number of processors used in this step is \((k\binom{(k+1)(n-1)}{k})\). Finally, select a set \(S^*\) from these \(\binom{(k+1)(n-1)}{k}\) sets such that \(w(MST(U_{k+1} - S^*))\) is maximized. The set \(S^*\) can be found in \(O(\log \binom{(k+1)(n-1)}{k})\) time using \((\binom{(k+1)(n-1)}{k})\) processors on an EREW PRAM. Therefore, there exists a simple parallel algorithm for this problem which requires \(O(k \log n \log \log n)\) time and \(O(kn\binom{(k+1)(n-1)}{k})\) processors on an EREW PRAM. When \(k\) is fixed, we have the following corollary.

**Corollary 6.1** The \(k\) MVE MST problem on \(G(V, E)\) with fixed \(k\) can be solved in \(O(\log n \log \log n)\) time using \(O(n^{k+1})\) processors on an EREW PRAM.

In the following we will show how to obtain a better algorithm for this problem. We first develop a simple sequential algorithm. Then we present an efficient sequential implementation of this algorithm. Finally we show how to parallelize the sequential algorithm in subsection 6.5.2.

### 6.5.1 A sequential algorithm and its efficient implementation

Recall that \(T\) is the MST of \(G\), and \(S^*\) is the set of \(k\) edges whose deletion results in the maximum increase in the weight of the MST of \(G(V, E - S^*)\). By Lemma 6.9, \(E(T) \cap S^* \neq \emptyset\). Let \(|E(T) \cap S^*| = r\). Obviously \(r \leq k\). In the following we construct an auxiliary undirected, weighted, multigraph \(G_0 = (V_0, E_0)\). (A graph is a multigraph if, for a pair of vertices \(u\) and \(v\), there exist multiple edges between them.) We later
show that the $k$ MVE MST problem on $G$ can be reduced to the $(k-r)$ MVE MST problem on $G_0$ with given $r$.

The construction of $G_0$ is as follows. Delete the $r$ edges in $E(T) \cap S^*$ from $T$. After that, $T$ becomes a forest $F$ of $r+1$ trees, $r \leq k$. For each tree in $F$, we use its root label to label all vertices in it. Now the vertex set $V_0$ of $G_0$ consists of all trees in $F$. Let $T_u$ and $T_v$ be two trees in $F$, an edge $(T_u, T_v) \in E_0$ with weight $w(e)$ if and only if there exists an edge $e = (u, v) \in E$ such that $u \in T_u$ and $v \in T_v$ and $T_u \neq T_v$. Obviously $G_0$ is a multigraph with $r+1$ vertices and $|E_0| \leq |E - E(T)| = m - (n - 1) < m$ edges.

From the graph $G_0$, we construct a subgraph $G_1(V_1, E_1)$ of $G_0$ as follows. Make $V_1 = V_0$. Let $T_u$ and $T_v$ be two vertices in $G_0$, sort the edges in $E_0$ which connect $T_u$ and $T_v$ in increasing order, using the weight associated with each edge as the key. The first $k-r+1$ edges are included in $E_1$. Note that the number of edges in $G_1$ is $|E_1| = O(r^2(k - r + 1)) = O(k^3)$ which is independent of $n$ and $m$.

**Lemma 6.13** Let $G_0$ and $G_1$ be defined as above. Then the $k'$ MVE MST problem on $G_0$ is equivalent to the $k'$ MVE MST problem on $G_1$, where $0 \leq k' \leq k-r$.

**Proof.** The approach we adopted is to show that the sparse, weighted $(k' + 1)$-edge certificates of both $G_0$ and $G_1$ are the same. Then from Lemma 6.2, this lemma follows. So, in the rest of this proof we show that the sparse, weighted $(k' + 1)$-edge certificates of both $G_0$ and $G_1$ are the same. Notice that $G_1$ is a subgraph of $G_0$.

Let $T^i_0$ be the MSF of $G_0$, and $T^i_j$ be the MSF of $G^j_0 = G_0 - \bigcup_{j=1}^{i-1} T^j_0$. By the definition in Section 6.2, the graph $G'' = \bigcup_{j=1}^{k' + 1} T^j_0$ is the sparse, weighted $(k' + 1)$-edge certificate of $G_0$. We observe that, for each pair of vertices $T_u$ and $T_v$ in $G''$, there are at most $k' + 1$ edges between them, and each of these $k' + 1$ edges belongs to a unique forest $T^i_0$, only, $1 \leq i \leq k' + 1$. Obviously these edges between $T_u$ and $T_v$ are contained in the first $k' + 1$ minimum weighted edges. By the definition of $G_1$, these $k' + 1$ edges are included in $G_1$. Therefore, $G''$ is a subgraph of $G_1$. Thus, the sparse, weighted $(k' + 1)$-edge certificate of $G_1$ is also that of $G''$. The lemma follows.

**Lemma 6.14** Given $T$, $S^*$, $F$ and $G_0$ as defined above, and $|E(T) \cap S^*| = r \neq 0$, the MST in $G(V, E - S^*)$ is equal to the union of the MST in $G_1(V_1, E_1 - S^*)$ and $F$.

**Proof.** Let $T^*$ be the MST of $G(V, E - S^*)$. First we show that any edge $e$ of $F$ must be contained in the MST of $G(V, E - S^*)$. Suppose $e$ is not, then adding $e$ to $T^*$ will form a cycle, and $w(e)$ must be larger than any other edges’ weight on the cycle, because otherwise we can obtain a smaller weighted spanning tree by replacing an edge $e'$ on $e$ if $w(e') > w(e)$. However, if $w(e)$ is larger than any other edges’ weight on the cycle, then $e$ should not have been included in $T -$ the MST of $G$ which leads to a contradiction. So, any edge in $F$ must be contained in the MST of $G(V, E - S^*)$.

Second, it is clear that any edge connecting two vertices of a tree in $F$ cannot be included in $T^*$. Therefore, $E(T^*) - E(F) \subseteq E(G_0)$. Let $G^*$ be the subgraph of $G_0$ induced by $E(T^*) - E(F)$. Obviously $G^*$ must be connected, otherwise the trees in $F$ will not be connected by the edges in $E(T^*) - E(F)$. Also, $G^*$ must be
acyclic, otherwise, a cycle in $T^*$ would occur. Therefore, $G^*$ must be a spanning tree of $G_0$. On the other hand, any spanning tree of $G_0$ together with $F$ will form a spanning tree of $G(V, E - S^*)$. Since $w(F)$ is fixed, $T^* = F \cup \text{MST}(G_0)$, and $w(T^*) = w(F) + w(\text{MST}(G_0))$. Since $\text{MST}(G_0) = \text{MST}(G_1)$, the lemma follows. □

By Lemmas 6.6, 6.8, and 6.9, we have the following recursive, sequential algorithm for the $k$ MVE MST problem. The input of the algorithm is a weighted, undirected graph $G$ ($G$ may be a multigraph), $n$ and $k$. The output is the set $S$ of $k$ edges and the maximized weight $W$ of the MST of $G(V, E - S)$.

**Procedure** Find_Vital_Edges($G, n, k, W, S$);

\[ MAX := -\infty; \; S^* := \emptyset; \]

if $k = 0$ then

return $W := w(\text{MST}(G)); \; S := \emptyset$;

else

compute $T$, the MST of $G$;

for $r := 1 \text{ to } k$ do

for each set $S_0$ of $r$ edges do

/* $S_0$ is selected from $E(T)$ systematically, one by one. */

delete the edges in $S_0$ from $T$;

construct the graph $G_1$;

Call Find_Vital_Edges($G_1, r + 1, k - r, W_1, S_1$);

if $w(T) - w(S_0) + W_1 > MAX$

then $S^* := S_0 \cup S_1$;

$MAX := w(T) - w(S_0) + W_1$

endif

endfor

$S := S^*; \; W := MAX$;

endif

Now we analyze the time complexity of this algorithm.

**Lemma** 6.15 The $k$ MVE MST problem on $G$ with fixed $k$ can be solved in time $O(n^k m \log n)$.

**Proof.** The time complexity of the algorithm above can be expressed by the following recursive equation.

\[
T_A(n, m, k) = \sum_{r=1}^{k} \binom{n-1}{r} [t_{\text{MST}(G)}(m, n) + T_A(r + 1, \frac{r(r + 1)}{2}(k - r + 1), k - r) + T_{G_1}(m, n)]
\]  

(6.6)

where
\[ T_A(n, m, k) \text{ is the time complexity of the above algorithm for the } k \text{ MVE MST problem on a graph (either a simple graph or a multigraph) with } n \text{ vertices and } m \text{ edges;} \]

\[ t_{\text{MST}(G)}(m, n) \text{ is the time complexity for constructing the MST of the graph } G \text{ with } n \text{ vertices and } m \text{ edges;} \]

\[ T_{G_1}(m, n) \text{ is the time complexity for constructing the graph } G_1 \text{ starting from a graph with } n \text{ vertices and } m \text{ edges.} \]

Obviously \( t_{\text{MST}(G)}(m, n) = O(m \log \beta(m, n)) \), \( T_{G_1}(m, n) = O(m \log n) \) because the construction of \( G_1 \) needs sorting all edges in \( G_0 \) which costs \( O(m \log n) \) time, and by the naive algorithm in Section 6.2, \( T_A(r + 1, \frac{r(r+1)}{2}(k - r + 1), k - r) \leq C \) where \( C \) is a constant. The reason is that \( G_1 \) contains constant vertices and edges if \( k \) is fixed. Thus

\[
T_A(n, m, k) = \sum_{r=1}^{k} \left( \frac{m^r}{r!} \right) [t_{\text{MST}(G)}(m, n) + T_A(r + 1, \frac{r(r+1)}{2}(k - r + 1), k - r) + T_{G_1}(m, n)] \leq (n + n^2 + \ldots + n^k)(m \log \beta(m, n) + C + m \log n) = O(n^k m \log n) \text{ with fixed } k. \quad \Box
\]

The algorithm above can be further improved by applying Lemma 6.2, i.e., we use the sparse, weighted \((k+1)\)-edge certificate \( U_{k+1} \) of \( G \) instead of \( G \) itself as the initial input. Meanwhile, we use the sparse, weighted \((k-r+1)\)-edge certificate \( G'' \) of \( G_1 \) instead of \( G_1 \). As a result, we derive the time complexity of Algorithm \textit{Find_Vital_Edges} is \( O(n^{k+1} \log n) \) with fixed \( k \). Though the time complexity of the algorithm in Lemma 6.15 is inferior to that in Equation 6.2, in the following we will present an efficient implementation of this algorithm to improve its time complexity. As a result, the algorithm can run in \( O(n^{k+1}) \) time when \( k \) is fixed. This result improves the results above as well as the result in [158]. Furthermore, this algorithm also leads to a better parallel algorithm for the \( k \) MVE MST problem in Section 6.5.2.

Now we present an efficient sequential implementation of the algorithm. Recall that if we already knew the specified \( r \) most vital edges in \( T \), by the arguments above, the other \( k - r \) most vital edges can be identified from the multigraph \( G_1 \). Since \( G_1 \) contains constant vertices and edges, by the naive algorithm in Section 6.2, we can easily identify the \( k - r \) most vital edges in \( G_1 \) in \( O(1) \) time. So how to construct the graph \( G_1 \) is the key. At this before, the construction of \( G_1 \) uses the sorting routine which costs \( O(|E(U_{k+1})| \log |E(U_{k+1})|) = O(n \log n) \) time. Here we show that there exists a better way to construct \( G_1 \), and this construction can be done in \( O(n) \) time. We proceed as follows. Let the \( r \) most vital edges in \( T \) be \( e_1, e_2, \ldots, e_r \). First, delete these \( r \) edges from \( T \). Then compute all CCs of \( T - \{e_1, e_2, \ldots, e_r\} \), and label all vertices in a CC with a unique identification, which can be implemented in \( O(n) \) time because the graph is a forest. Finally, let us consider the graph \( G_1 \) which contains \( r + 1 \) vertices, each one corresponds to a CC. There are \( O((r+1)^2) = O(k^2) \) vertex pairs in \( G_1 \). For each pair of vertices in \( G_1 \), for example \( C_1 \) and \( C_2 \), we compute the edges between them by the following procedure. Let \( R \) be all edges in \( U_{k+1} \). We delete all other edges whose endpoints are not labeled by \( C_1 \) and \( C_2 \) respectively. Let the remaining edge set
be $R'$. Obviously, $R'$ can be obtained in $O(n)$ time because $|R'| \leq |R| = O(n)$. Now select the $(k - r) + 1$ smallest element from $R'$ using the weight associated with each edge as the searching key. Assuming that $e$ is the $(k - r) + 1$ smallest element of $R'$. Compare all other elements $e'$ in $R'$ with $e$, if $w(e') \leq w(e)$, $e'$ is kept in $R'$; otherwise $e'$ is deleted from $R'$. Let $R''$ be a subset of $R'$ which contains all the remaining elements in $R'$. Then, by the definition, $R''$ is the edge set between vertices $C_1$ and $C_2$ in $G_1$. $R''$ can be obtained in $O(n)$ time because selecting an element from a set of $O(n)$ elements costs $O(n)$ time [2]. Therefore, the construction of $G_1$ requires $O(k^2n) = O(n)$ time because $G_1$ has $O(k^2)$ vertex pairs. Having the graph $G_1$, the $k - r$ most vital edges in $G_1$ can be identified in constant time, and the weight, denote by $w(G_1, k - r)$, of the MST in the remaining graph of $G_1$ by deleting these $k - r$ edges from $G_1$ can be obtained in constant time. Therefore, the weight of the remaining graph induced on $G$ by deleting these $k$ most vital edges from $G$ can be derived, i.e., the weight is $w(MST(G)) - \sum_{i=1}^{r} w(e_i) + w(G_1, k - r)$ which can be obtained in constant time. Now we state the following theorem.

**Theorem 6.8** The $k$ MVE MST problem on a weighted, connected, undirected simple graph $G = (V, E)$ can be solved in time $O(n^{k+1})$ with fixed $k \geq 1$.

**Proof.** The MST of $G$ can be obtained in time $O(m \log \beta(m, n)) = O(n^2)$ by the algorithm in [63], and the $U_k$ can be achieved in the same time bound due to $k$ is constant. There are $\binom{n-1}{r} = O(n^r)$ ways to choose $r$ edges from $T$. For each specified $r$ edges, if each of them is among the $k$ most vital edges in $G$, the other $k - r$ most vital edges in $G$ can be identified in $O(n)$ time by $G_1$ because the construction of $G_1$ needs $O(n)$ time. The weight of the MST in the remaining graph after deleting these $r + (k - r)$ edges from $G$ can be obtained in $O(n)$ time because of the same reason. Let $S$ be such a set of the potential $k$ most vital edges. Since $1 \leq r \leq k$, we need to choose one $S_0$ of $k$ edges from a total $\sum_{r=1}^{k} \binom{n-1}{r} = O(n^k)$ sets of $S$ such that the MST in the remaining graph $G(V, E - S_0)$ has the maximum weight. By the discussion above, for each of these sets of $S$, it needs $O(n)$ time to compute the weight of the MST in the remaining graph $G(V, E - S)$. So, we need $O(n^{k+1})$ time to solve the $k$ MVE MST problem. □

**6.5.2 A parallel implementation**

Assume that the edges in $T$ have been numbered from 1 to $n - 1$. The parallelization of Algorithm $\text{Find\_Vital\_Edges}$ is not difficult. We make $O(n^k)$ copies of $T$. This can be implemented by applying a broadcasting technique in which a single value is broadcast to $O(n^k)$ places using prefix computation. Thus, this step needs $O(k \log n) = O(\log n)$ time and $O(n^{k+1})$ processors on an EREW PRAM because there are $n - 1$ edges in $T$. Then, there are a total of $\binom{n-1}{r}$ subsets of $r$ edges in $T$ for different $r$ with $1 \leq r \leq k$, and each such a subset is numbered. For a specific subset $S$ numbered $i$, there exists a corresponding copy of $T$, we delete these $r$ edges in $S$ from this copy of $T$. As a result, we obtain a spanning forest $F$. For every vertex in $F$, compute its
CC identification. Here we label each vertex with the root’s label of the tree to which it belongs. This can be implemented by applying the tree contraction algorithm of Abrahamson et al. [1] to \( F \) which requires \( O(\log n) \) time using \( O(n/\log n) \) processors on an EREW PRAM. Having done the above, it is easy to construct \( G_0(V_0, E_0) \). Obviously \(|E_0| \leq |E| - (n - 1) - |S| < m\). Now we give the details of constructing the graph \( G_1 \). Suppose that \( V = \{1, 2, \ldots, n\} \), and every undirected edge \((u, v)\) in \( E_0 \) is stored in the form \((\min\{u, v\}, \max\{u, v\})\). We label each endpoint of the edges in \( E_0 \) by its CC identification. Sort all edges in \( E_0 \) in increasing order, using the labels of the endpoints of each edge as the primary key, and the weight associated with the edge as the second key. Let the sorted sequence be \( R \). Define a subsequence \( R' \) of \( R \) as an interval if the endpoints of all edges in \( R' \) have the same labels. Delete all the other edges in \( R' \) except the first \( k - r + 1 \) edges. This can be implemented by compressing \( R \) using prefix computation. The remaining edges in the compressed \( R \) are the edges of \( G_1 \). Sorting and prefix computation can be done in \( O(\log n) \) time using \( O(m) \) processors on an EREW PRAM [30] because there are \( O(m) \) edges to be sorted. Note that \( G_1 \) is a multigraph with \( k + 1 \) vertices and \( ck^3 \) edges at most, and \( ck^3 \leq m \), where \( c \) is a constant. Then, find the sparse, weighted \((k - r + 1)\)-edge certificate \( G'' \) of \( G_1 \) which requires \( O(k \log k \log \log k) \) time using \( O(k^3) \) processors on an EREW PRAM because \( G_1 \) contains \( O(k^3) \) edges at most. It is easy to derive that the depth of recursion of the algorithm is bounded by \( O(k) \). When \( k \) is fixed, we have

**Lemma 6.16** Given the MST of \( G \), the \( k \) MVE MST problem on \( G \) with fixed \( k \) can be solved in \( O(\log n) \) time using \( O(mn^k) \) processors on an EREW PRAM.

**Proof.** For a fixed \( r \), there are a total of \( \binom{n - 1}{r} \) different ways to select \( r \) edges from \( T \). For a given specific \( r \) edges in \( T \), constructing \( G_i \) requires \( O(\log n) \) time and \( O(m) \) processors on an EREW PRAM, \( i = 0, 1 \). The \((k - r)\) MVE MST problem on \( G_1 \) can be solved in constant time because \( G_1 \) contains \( O(k) \) vertices and \( O(k^3) \) edges and \( k \) is fixed. Since \( r \) is bounded by \( 1 \leq r \leq k \), there are a total of \( mP(n, r) = m \sum_{r=1}^{k} \binom{n-1}{r} = O(mn^k) \) processors required. The lemma follows. \( \square \)

One direct application of Lemma 6.16 is to find the \( k \) most vital edges in a graph \( G \) with bounded degree \( d \). For this special case, the number of edges of \( G \) is \( m = O(n) \). Thus, we have

**Lemma 6.17** Given a weighted, undirected, simple graph \( G(V, E) \) with bounded degree \( d \) and the MST of \( G \), the \( k \) MVE MST problem on \( G \) can be solved in \( O(\log n) \) time using \( O(n^{k+1}) \) processors on an EREW PRAM.

Combining Lemmas 6.16 and 6.17, we have the following theorem immediately.

**Theorem 6.9** Given a weighted, undirected, connected simple graph \( G(V, E) \), the MST of \( G \), and the sparse, weighted \((k + 1)\)-edge certificate of \( G \), the \( k \) MVE MST problem on \( G \) with fixed \( k \) can be solved in \( O(\log n) \) time using \( O(n^{k+1}) \) processors on an EREW PRAM.
Proof. We use $U_{k+1}$ instead of $G$ as the initial input of the algorithm. Note that $m = |E(U_{k+1})| \leq (k + 1)(n - 1) = O(n)$, since $k$ is fixed. By Lemma 6.16, the theorem follows. □

Note that Theorem 6.9 holds only when $T$ and $U_{k+1}$ are given. If not so, the preprocessing for the construction of $U_{k+1}$ requires $O(\log n \log \log n)$ time using $O(m + n)$ processors on an EREW PRAM.

We now summarize all known results for solving the $k$ MVE MST problem by Table 6.1.

6.6 Conclusions

In this chapter we developed a better exact algorithm for the $k$ MVE MST problem for general $k$. A non-trivial part is to use the sparse, weighted $(k + 1)$-edge certificate of a weighted undirected graph to guide the algorithm design. We presented efficient sequential and parallel algorithms for this problem when $k > 3$ is fixed. Particularly we presented the most efficient NC algorithms so far on an EREW PRAM and/or a CREW PRAM when $k = 1, 2, 3$. 
<table>
<thead>
<tr>
<th>Authors</th>
<th>Sequential</th>
<th>EREW</th>
<th>CREW</th>
<th>CRCW</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hsu et al [83]</td>
<td>$O(n^2)$ or $O(m \log n)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hsu et al [88]</td>
<td>$O(t_{MST} + \min { m \alpha(m,n), m + n \log n })$</td>
<td>$T = O(n^{1+x})$; $P = O(n^{1-x})$, $0 &lt; x &lt; 1$</td>
<td></td>
<td></td>
<td>$k = 1$</td>
</tr>
<tr>
<td>Iwano et al [93]</td>
<td>$O(t_{MST} + \min { m \alpha(m,n), m + n \log n })$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This chapter</td>
<td>$O(n^2 \alpha(3n,n))$</td>
<td>$T = O(\log n \log \log n)$, $P = O(n) + \frac{n^2}{\log n \log \log n}$</td>
<td>$T = O(\log n \log \log n)$, $P = O(n)$</td>
<td></td>
<td>$k = 1$</td>
</tr>
<tr>
<td>This chapter</td>
<td>$O(n^3 \alpha(4n,n))$</td>
<td></td>
<td></td>
<td>$T = O(\log n \log \log n)$, $P = O(n^3/\log n)$</td>
<td>$k = 3$</td>
</tr>
<tr>
<td>This chapter and $k$ is fixed</td>
<td>$O(n^{k+1})$</td>
<td>$T = O(\log n \log \log n)$, $P = O(n^{k+1})$</td>
<td></td>
<td></td>
<td>$k \geq 4$</td>
</tr>
<tr>
<td>Shen [157]</td>
<td></td>
<td></td>
<td></td>
<td>$T = O(\log n)$, $P = O(m \log \log \log n / \log n + n)$</td>
<td>$k = 1$</td>
</tr>
<tr>
<td>Shen [158] and $k$ is fixed</td>
<td>$O(n^k m \log \beta(m,n))$</td>
<td></td>
<td></td>
<td></td>
<td>$k \geq 2$</td>
</tr>
</tbody>
</table>

Table 6.1: A comparison of the results
Chapter 7

Finding the Single Most Vital Edge on Mesh and Hypercube Processor Arrays

7.1 Introduction

Let $G(V, E, w)$ be an undirected, weighted graph with a non-negative weight $w(e)$ associated with each edge $e$ in $E$. Define the single most vital edge with respect to (w.r.t.) a minimum optimization problem as follows. Let $P$ be an instance of such a minimum optimization problem on $G$, and $e^*$ be an edge of $G$, $e^*$ is the single most vital edge of $G$ if and only if whose removal from $G$ maximizes the value of this minimum optimization problem in the remaining graph $G' = G(V, E - \{e^*\}, w)$.

For the single most vital edge problem w.r.t. MSTs, it has been extensively studied in the past couple years. A detailed survey can be found in Chapter 6. Also in that chapter, we already presented efficient parallel algorithms for the problem based on PRAMs. In this chapter we will show how to solve the problem based on interconnection networks such as meshes and hypercubes. Here we should point out that Hsu et al. [88] once presented an $O(n^2)$ work parallel algorithm for this problem on a hypercube model. Their algorithm needs $O(n^{1+x})$ time using $O(n^{1-x})$ processors with the assumption that each processor has size of $f(n) = O(n)$ memory units, $0 < x < 1$.

For the single most vital edge problem w.r.t. shortest paths (SPs) between two specified vertices $s$ and $t$, a number of researchers have made some contributions to it in the past decade. Corley et al. [33] first raised this problem by giving some preliminary results. Malik et al. [130] later presented an efficient algorithm when $G$ is an undirected graph. Unfortunately, their algorithm is incorrect, as shown by Bar-Noy et al. [16] who gave a counter-example. The generalization of this problem is the $k$ most vital edge problem w.r.t. SPs between $s$ and $t$ (the $k$ MVE SP problem for short) [15, 16, 33, 124]. Corley et al. stated in [33] that the identification of the sufficient conditions would lead to a constructive procedure to solve the $k$ MVE SP problem. Ball et al. [15] generalized this problem by assigning a removal cost to each edge, and the total removal cost is bounded. For this general case, they proved that the problem is NP-complete. However, their proof does not imply that the $k$ MVE SP problem is also NP-complete. Bar-Noy
et al. [16] have shown it is NP-complete indeed. Recently, Liang et al. [124] gave a fast polynomial algorithm for this problem when \( k \) is fixed.

In this chapter we will consider the single most vital edge problem w.r.t. either MSTs or SPs between two specified vertices \( s \) and \( t \) by presenting fast algorithms for them on mesh-connected and hypercube-connected processor arrays respectively. The proposed algorithms for these two problems need \( O(n) \) time on an \( n \times n \) mesh-connected processor array, which are also time-optimal because of a trivial lower bound for global communication on this model needs \( \Omega(n) \) time. By designing algorithms on an \( n \times n \) mesh, it is interesting to find that, on this model, the single most vital edge problem w.r.t. SPs can be reduced into a single most vital edge problem w.r.t. MSTs. Moreover, we also find some efficient sequential and parallel algorithms (on the PRAMs) for the above problems are inappropriate for this special model. Instead, some inefficient sequential algorithms have efficient implementations on the model. The algorithms on an \( n \times n \times n \) hypercube-connected processor array need \( O(\log^2 n) \) time which are the fastest, compared with a previously known result [88]. The algorithms on the hypercube array are the simulations of the algorithms on the mesh-connected processor array. Though both Hsu et al. [88] and us use the hypercube as the computational model, there is a little difference in the definition of the model. That is, they allow each processor has size of \( f(n) \) local memory where \( f(n) \) is a function of \( n \), but we only allow each processor has a constant registers (constant memories). As a result, their algorithm needs \( O(n^{1+\varepsilon}) \geq \Omega(n) \) time, but our algorithm for the same problem needs \( O(\log^2 n) \) time only. It is not difficult to see that our better time bound is achieved by making use of more processors.

Without loss of generality, we assume that \( G(V,E,w) \) is 2-edge connected when dealing with the MST problem, or there are at least two edge-disjoint paths between \( s \) and \( t \) when dealing with the SP problem. Otherwise, when we delete an edge which is a bridge of \( G \) or the unique edge between \( s \) and \( t \), the remaining graph is disconnected, hence no an MST or a shortest path between \( s \) and \( t \) exists.

The rest of this chapter is organized as follows. In Section 7.2 we define the computational models. The algorithms for the single most vital edge problem w.r.t. both MSTs and SPs between \( s \) and \( t \) on an \( n \times n \) mesh array are presented in Section 7.3, and the algorithms for these two problems on an \( n \times n \times n \) hypercube array are given in Section 7.4. A conclusion is given in Section 7.5.

### 7.2 The Computational Models

#### 7.2.1 The 2-D mesh array

We use the model described by Atallah et al. [7]. The computational model consists of \( N = n \times n \) identical processors positioned on a square array, and the processor positioned at \((i,j)\) is represented by \( PE_{i,j} \), where \( 0 \leq i,j \leq n - 1 \), i.e., the processors are arranged in \( n \) rows and \( n \) columns. Each processor can communicate with its four neighbors provided they exist. A processor has a constant number of registers, each of which can store a word of \( c \log n \) bits for some constant \( c > 0 \). A processor can
do the usual Arithmetic and Boolean operations. There is a single instruction stream:
Within a unit time, the same instruction is broadcast to all processors, which execute it and wait for the next one. A single instruction at a processor can consist of an arithmetic operation, transmitting a word to all neighbors and receiving a word from its neighbors. For the discussion convenience later, we refer to this model as 2-D mesh array.

7.2.2 The hypercube array

Assume that \( p = 2^q \). Let \( i_{q-1}i_{q-2}\ldots i_0 \) be the binary representation of \( i \), \( 0 \leq i < p \), and \( i^{(b)} \) be the number whose binary representation is \( i_{q-1}i_{q-2}\ldots i_{b+1}i_{b}i_{b-1}\ldots i_0 \), where \( i_b \) is the complement of \( i_b \) and \( 0 \leq b < q \). In the hypercube model, a processor indexed \( i \) is directly connected to a processor indexed \( i^{(b)} \), \( 0 \leq b < q \). Obviously, each processor has \( q \) neighbors in a hypercube consisting of \( p \) processors. We later use \( X[i, j], Y[i, j] \) to represent the registers in \( PE_{i,j} \), \( 0 \leq i, j < n \).

In this chapter we consider the following hypercube model, and refer to it as hypercube array. The hypercube array consists of \( N = n^3 \) processors. Conceptually, these processors may be regarded as arranged in an \( n \times n \times n \) array pattern. Assume that the processors are indexed in row-major mode, i.e., the processor positioned at \((i, j, k)\), denoted by \( PE_{i,j,k} \), has an index \( in^2 + jn + k \). Thus, if \( r_{3q-1}\ldots r_2r_{q-1}\ldots r_0 \) is the binary representation of the index of \( PE_{i,j,k} \), then \( i = r_{3q-1}r_{3q-2}\ldots r_{2q}, j = r_{2q-1}r_{2q-2}\ldots r_q \), and \( k = r_{q-1}r_{q-2}\ldots r_0 \). We further impose the same restrictions on this model as that imposed on the 2-D mesh array. That is, each processor has only constant registers and each register can store a word of \( c \log n \) bits where \( c \) is constant. The Arithmetic operations and Boolean operations in all processors are synchronized, etc. We later use \( A[i, j, k], B[i, j, k] \) and \( C[i, j, k] \) to represent the registers in \( PE_{i,j,k} \), \( 0 \leq i, j, k < n \).

7.3 Algorithms on the 2-D Mesh Array

7.3.1 Finding the single most vital edge w.r.t. the MST problem

Referred to the NC algorithm for this problem in Section 6.3 of Chapter 6, in the following we present an implementation of the algorithm on the 2-D mesh array. Before we proceed, we first describe a few data movements on this model that will be used repeatedly throughout this subsection. We then present the implementation details of the algorithms step by step.

1. Horizontal Rotation \([7]\). Every processor creates a “duplicate” copy of its content; then all the duplicate data items move like a row of soldiers: every row moves left, its leftmost elements “bouncing back” on its leftmost processor \( PE_{i,0} \). The data that bounced in \( PE_{i,0} \) move right until they bounce again, this time on the rightmost processor \( PE_{i,n-1} \); then they move left again until they go back to their initial position. It is clear that the data initially in every \( PE_{i,j} \) visit all the other processors of row \( i \), and this takes \( O(n) \) time.
2. Vertical Rotation [7]. This is the vertical analog of the previous data movement. This operation obviously takes \( O(n) \) time, and the data initially in \( PE_{i,j} \) visit all the other processors in column \( j \).

Using Horizontal Rotation and Vertical Rotation, Atallah et al. [7] have shown that all pairs shortest paths and the transitive closure of \( G(V, E, w) \) can be computed in \( O(n) \) time on this model. Here we briefly describe their result for later use. Let \( A \) and \( A^* \) be the adjacency matrix and the transitive closure matrix of \( G \). Assume that an entry \( a_{i,j} \) of \( A \) is stored in \( PE_{i,j} \), and an entry \( a^*_{i,j} \) of \( A^* \) is also stored in \( PE_{i,j} \) for all \( i, j \), \( 0 \leq i, j \leq n - 1 \). Then,

**Lemma 7.1** [7] Let \( A \) be the adjacency matrix of \( G(V, E, w) \). The transitive closure matrix \( A^* \) of \( G \) can be computed in time \( O(n) \) on a 2-D mesh array.

Similarly, all pairs shortest paths of \( G \) can also be computed in \( O(n) \) time on this model.

3. Random Access Write [140]. In this formulation, an index \( (i', j') \) is contained in \( PE_{i,j} \). Data from a specified register of \( PE_{i,j} \) is to be transmitted to \( PE_{i',j'} \), \( 0 \leq i, i', j, j' \leq n - 1 \). It is assumed that no two processors are sending their data to the same destination, and if the destination address is \( \infty \), then the data from \( PE_{i,j} \) is not transmitted to any other processor. The details of how this data movement is implemented in \( O(n) \) time can be seen in [140].

4. Horizontal Broadcasting. Assume that in each row \( i \), an element \( a_i \) is stored in the register \( X \) of some processor, the objective to broadcast \( a_i \) to the register \( X \) of all other processors in row \( i \) for all \( i, 0 \leq i \leq n - 1 \). The procedure consists of several rounds. Initially, set the processor holding the element a special “tag:=1” in each row. Then, in each round every processor first sends the content of \( X \) and its tag to its right neighbor provided this right neighbor exists. Each receiver then checks whether its tag is “1”. If yes, it already received the data and did nothing. Otherwise, it checks whether the tag of the sender is “1”, if yes, copy the data to its register \( X \) and sets its tag by “1”. After finishing the first \( n - 1 \) rounds, we start by another \( n - 1 \) rounds, i.e., in each round each processor sends its contents of \( X \) and tag to its left neighbor, the rest operations are the same as the above and are omitted. Finally the element in some processor in row \( i \) is broadcast to all other processors of that row. The time used is \( O(n) \).

5. Vertical Broadcasting. This is the vertical analog of Horizontal Broadcasting. It obviously takes \( O(n) \) time. The data initially in \( PE_{i,0} \) is sent to all the other processors of column \( j \).

6. Horizontal Prefix. In this formulation, each processor \( PE_{i,j} \) contains an element \( d_{i,j} \) which is stored in the register \( D \). Then the elements in row \( i \) form a sequence \( d_{i,0}, d_{i,1}, \ldots, d_{i,n-1} \). The problem is to compute all prefixes of this sequence, i.e., the processor \( PE_{i,j} \) will contain the result \( \sum_{k=0}^{j} d_{i,k} \) which is stored in the register \( S[i,j] \) for all \( i, 0 \leq i \leq n - 1 \). We proceed as follows. First, all elements in column 0 set a tag with “1”, and the register \( S[i,j] \) in \( PE_{i,j} \) is assigned by \( S[i,j] := d_{i,j} \) initially. Then there are \( n - 1 \) rounds to run the following routine. Each processor sends the content of \( S \) and the tag to its right neighbor provided that the right neighbor exists. If the tag
of the sender is “1”, then the sender sets its tag by “0” after finishing sending. When a processor receives a message with the sender’s tag “1”, it sets \( S[i,j] := S[i,j] + s \) and sets its tag by “1”, where \( s \) is the value it just received. Obviously, the operations can be done in \( O(n) \) time.

7. Vertical Prefix. This is the vertical analog of Horizontal Prefix, omitted.

Having the subroutines above, we present the following lemmas.

**Lemma 7.2** Let \( a_0, a_1, \ldots, a_{n-1} \) be stored in \( n \) different processors on a 2-D mesh array. Along with \( a_i \), \( 0 \leq i \leq n-1 \), the destination processor address is also stored in the same processor. Assume that all destinations are distinct. Then routing these elements to their destinations needs \( O(n) \) time.

**Proof.** First, processor \( PE_{i',j'} \) forms a routing record “\( (PE_{i',j'}, PE_{\text{des}(a_i)}, a_i) \)” if it contains \( a_i \); otherwise forms a routing record “\( (PE_{i',j'}, \infty, 0) \)”, where \( PE_{\text{des}(a_i)} \) is the destination of \( a_i \). Then route these records to their destinations by Random Access Write. After that, \( a_i \) is sent to the \( PE_{\text{des}(a_i)} \) for all \( i, 0 \leq i \leq n-1 \). Routing takes \( O(n) \) time. The lemma thus follows. \( \square \)

**Lemma 7.3** Let \( a_0, a_1, \ldots, a_{n-1} \) be stored in some row (column) \( i \) on a 2-D mesh array, where \( a_i \) is stored in \( PE_{i,j} \). Broadcasting \( a_i \) to row \( i \) and column \( i \) needs \( O(n) \) time for all \( i, j, 0 \leq i, j \leq n-1 \).

**Proof.** First, each processor \( PE_{i,j} \) forms a routing record “\( (PE_{i,j}, PE_{i,i}, a_i) \)” if \( a_i \) is stored in it, otherwise forms a routing record “\( (PE_{i,j}, \infty, 0) \)”. Then route the records to their destinations by Random Access Write. After that, \( a_i \) is sent to the \( PE_{i,i} \). Finally the processors in each row (column) duplicate \( a_i \) to all processors by Horizontal Broadcasting (Vertical Broadcasting). Routing and broadcasting need \( O(n) \) time respectively. The lemma thus follows. \( \square \)

Now we begin to consider the 1 MVE MST problem by giving the implementation details. Assume that the weighted adjacency matrix \( A \) of \( G(V,E,w) \) is already stored in the 2-D mesh array, i.e., an entry \( a_{i,j} \) of \( A \) is stored in the register \( A[i,j] \) of \( PE_{i,j} \) for all \( i, j, 0 \leq i, j \leq n-1 \).

**Step 1.** Compute the MST of \( G(V,E,w) \). Let \( T \) be the MST of \( G \) which is stored in an adjacency matrix \( T \) where an entry \( T[i,j] = T[j,i] = 1 \) if \( (i,j) \) is an edge in \( T \), \( T[i,j] = T[j,i] = 0 \) otherwise. This step can be implemented in \( O(n) \) time by Maggs et al’s algorithm [129].

**Remark.** Atallah et al. [7] also suggested an algorithm for the MST problem on this model which has the same time bound as above. However, as said in [129] their algorithm is more complicated, for example, they achieved the \( O(n) \) time complexity by reducing the fraction of the mesh in use by a constant factor at each recursive call.

**Step 2.** Make \( T \) become a rooted directed tree \( T_{in} \) in which the direction of each edge is pointed to the root. We first select a vertex as the root of \( T \) arbitrarily, for example, vertex 0. Then compute the parent \( F(v) \) of \( v \) in \( T_{in} \) for all \( v \in V, v \neq 0 \).
Meanwhile, we construct another directed tree $T_{out}$ rooted at vertex 0 from $T$ in which the direction of each edge leaves from the root.

**Step 2.1.** Compute all pairs shortest paths of $T$ using the adjacency matrix of $T$. Let $T^*$ be the matrix of all pairs shortest paths in $T$, i.e., an entry $T^*[i, j]$ stores the distance between $i$ and $j$ in $T$. This substep can be implemented in $O(n)$ time by Lemma 7.1.

**Step 2.2.** Broadcast the value of $T^*[0, i]$ to the register $D_1[i, j]$ of $PE_{i,j}$ in row $i$ and to the register $D_2[j, i]$ of $PE_{j,i}$ in column $i$ for all $i, j, 0 \leq i, j \leq n - 1$. This can be done in $O(n)$ time by Lemma 7.2. Note that $T^*[0, i]$ is the level number of vertex $i$ in $T$ rooted at vertex 0.

**Step 2.3.** Construct the tree $T_{in}$. Every $PE_{i,j}$ runs the following instruction: if $T[i, j] = 1$ and $D_1[i, j] = D_2[i, j] - 1$, then $i$ is the parent of $j$ in $T_{in}$, set $T_{in}[i, j] := 1$ and $F(j) := i$. Otherwise, $T_{in}[i, j] := 0$. Note that the directed tree $T_{out}$ rooted at vertex 0 can be constructed similarly. This substep can be done in $O(1)$ time clearly.

**Step 3.** Check whether some edges of $T_{in}$ are the bridges of $G$, using the algorithm in [7]. If yes, then any one of such edges is the single most vital edge of $G$, stop. Otherwise we continue the following steps. This step can be done in $O(n)$ time [7].

**Step 4.** Given $T_{in}$, compute the pre-order numbering $pre(v)$, the number of ancestors $na(v)$, and the number of descendants $nd(v)$ for every vertex $v \in V$.

**Step 4.1.** Compute $nd(v)$ for each $v \in V$. We proceed as follows. First, compute the transitive closure $T_{out}^*$ of $T_{out}$ using the adjacency matrix of $T_{out}^*$. Then for each row $i, 0 \leq i \leq n - 1$, compute $\sum_{j=0}^{n-1} T_{out}^*[i, j]$, and store this value to the register $ND[i, 0]$ of $PE_{i,0}$. It is clear that $nd(i) = \sum_{j=0}^{n-1} T_{out}^*[i, j] + 1$ because $i$ is a descendant of itself by our definition. This substep can be done in time $O(n)$.

**Step 4.2.** Similarly, using the adjacency matrix of $T_{in}$, we can compute the transitive closure $T_{in}^*$ of $T_{in}$. As a result, $na(i) = \sum_{j=0}^{n-1} T_{in}^*[i, j] + 1$ because $i$ is an ancestor of itself by our definition. Assume that $na(i)$ is stored in the register $NA[i, 0]$ of $PE_{i,0}$. 

**Step 4.3.** Broadcast $NA[i, 0]$ to the register $NA_1[i, j] := NA[i, 0]$ of $PE_{i, j}$ in row $i$ and to the register $NA_2[j, i] := NA[i, 0]$ of $PE_{j,i}$ in column $i$; broadcast $ND[i, 0]$ to the register $ND_1[i, j] := ND[i, 0]$ of $PE_{i,j}$ in row $i$ and to the register $ND_2[j, i] := ND[i, 0]$ of $PE_{j,i}$ in column $i$, for all $i, j, 0 \leq i, j \leq n - 1$. This can be done in time $O(n)$ by Lemma 7.3.

Before we continue, we define an ordered tree $T(V, E')$ as follows. $T$ is a tree rooted at $r$. For each other vertex $v \neq r$, there exists another vertex $F(v)$ which is the parent of $v$ in $T$, and for a parent vertex $u$, all $u$‘s children $v_1, v_2, \ldots, v_l$ are ordered, i.e., $v_i$ is the $i$th child of $u$, where $F(v_i) = u$ for $1 \leq i \leq l$. Having this definition, Tsin and Chin [166] have given the following lemma.

**Lemma 7.4** [166] Let $T(V, E')$ be an ordered tree. For each $i \in V$,

$$pre(i) = \sum_{j \in ANC(i)} \sum_{w \in EBRO(j)} nd(w) + na(i),$$

(7.1)

where $ANC(i)$ is the set of all ancestors of $i$, $EBRO(j)$ is the set of all elder brothers of $j$, $na(i)$ is the number of ancestors of $i$. 
Step 4.4. Make $T_{in}$ become an ordered tree. Note that in row $i$, if $T_{in}[i, j] = 1$, then $j$ is one of children of $i$. Based on this fact we assign the ranks (orders) to the children of $i$. First each $PE_{i,j}$ creates a temper register $Temp[i, j]$, and executes the following operation:

$$\text{if } T_{in}[i, j] = 1 \text{ then } Temp[i, j] := 1 \text{ else } Temp[i, j] := 0.$$ 

Then compute all prefixes for the sequence $Temp[i, 0], Temp[i, 1], \ldots, Temp[i, n-1]$ for each row $i, 0 \leq i \leq n - 1$. $Fix[i, j] := \sum_{k=0}^{j} Temp[i, k]$ for $i, 0 \leq i \leq n - 1$. Now we assign the ranks to the children of $i$. For each $PE_{i,j}$, if $T_{in}[i, j] = 1$, then $j$ is the $Fix[i, j]$th child of $i$. The rank of $j$ is stored in the register $Rank[i, j]$ by setting $Rank[i, j] := Fix[i, j]$ by $PE_{i,j}$. Otherwise $Rank[i, j] := 0$. $T_{in}$ now is an ordered tree already. The rest is to compute $pre(v)$ for each $v \in V$ in $T_{in}$.

Step 4.5. By Equation 7.1, we first compute $\sum_{w \in EBR0(i)} nd(w)$ for every $j$. Each processor $PE_{i,j}$ executes the following assignment: $Temp[i, j] := ND_0[i, j]$ (the number of descendants of $j$ in $T_{in}$) if $T_{in}[i, j] = 1$; otherwise $Temp[i, j] := 0$. Then compute all prefixes of the sequence $Temp$ for each row $i$. Let $Fix[i, j] := \sum_{k=0}^{j} Temp[i, k] - Temp[i, j]$. Obviously, $Fix[i, j] = \sum_{w \in EBR0(j)} nd(w)$. Now each $PE_{i,j}$ sets $Ebro[i, j] := Fix[i, j]$ if $T_{in}[i, j] = 1$; sets $Ebro[i, j] := 0$ otherwise.

Step 4.6. Compute $\sum_{j \in EBR0(i)} \sum_{w \in EBR0(j)} nd(w)$ for every $i$. First, each $PE_{i,j}$ runs the following instruction: $Temp[i, j] := Ebro[i, j]$ if $T_{in}[i, j] = 1$ (is an ancestor of $j$); otherwise $Temp[i, j] := 0$. Then all processors in row $i$ compute all prefixes of the sequence $Temp$ at row $i$. $Fix[i, j] := \sum_{k=0}^{j} Temp[i, k]$. Obviously $Fix[i, n-1] = \sum_{j \in EBR0(i)} \sum_{w \in EBR0(j)} nd(w)$ for all $i, 0 \leq i \leq n - 1$. Third, the value of $Fix[i, n-1]$ is sent to the register $PRE[i, 0]$ of $PE_{i,0}$ by Random Access Write. Finally, $PE_{i,0}$ sets $PRE[i, 0] := PRE[i, 0] + NA[i, 0]$, i.e., $pre(i) = PRE[i, 0]$ by Equation 7.1. Then, we have the following lemma.

Lemma 7.5 Given an undirected tree $T(V, E')$ and a specified vertex $v$ as the root, computing $pre(v)$, $na(v)$, and $nd(v)$ for each vertex $v \in V$ can be done in $O(n)$ time on the 2-D mesh array.

Remarks. Atallah et al. [6] suggested $O(\sqrt{n})$ time algorithms on a $\sqrt{n} \times \sqrt{n}$ mesh for a class of tree problems with $n$ vertices including computing pre-order numbering, post-order numbering, the height (level) of the tree, the number of descendants, etc. From there we know that their algorithms for these problems have better performances than ours. However, due to the structure of the 2-D mesh array, to solve all non-trivial problems on this model needs $\Omega(n)$ time. So, even using their algorithms as a subroutine to solve our problem, it still needs $\Omega(n)$ time at least. Furthermore, their algorithm has the following disadvantages: (i) all of their algorithms make use of linked prefix computation as a subroutine whose implementation on this model is quite complicated, compared with our algorithm which utilizes the transitive closure computation as a subroutine that is very efficient on this model; (ii) their algorithm requires that the processors are indexed by the shuffled row-major mode. Since this indexing scheme is not consistent with the indexes of elements in the adjacency matrix.
of a graph, it makes the mapping between the elements in an adjacency matrix and the processors in the 2-D mesh array more difficult.

Step 5. Broadcast pre(i) to the register PRE1[i,j] of PEi,j in row i, and to the register PRE2[j,i] of PEj,i in column i for all i,j, 0 ≤ i, j ≤ n – 1.

Step 6. Construct T2 — the MST of the graph G(V,E – E(T),w) using Maggs et al’s algorithm. The weighted adjacency matrix A’ of this graph can be obtained in O(1) time by checking whether an edge (i,j) is in T. If it does, set A’[i,j] := 0. Otherwise, A’[i,j] := A[i,j], where A is the weighted adjacency matrix of G(V,E,w). This step can be implemented in time O(n). Having the tree T2, a directed version T2’ of T2 is constructed, in which the direction of each edge is pointed to the root.

Step 7. Construct a routing record for every PEi,j. PEi,j forms a record “(i,j,pre(i), pre(j), w(i,j), PE0,j)” if (i,j) is an edge of T2 and i is the parent of j; otherwise forms a record “(i,j,pre(i),pre(j), w(i,j), ∞)”, where pre(i) = PRE1[i,j] and pre(j) = PRE2[i,j] are already available in PEi,j, and PE0,j is the address of the destination processor. Routing the records to their destinations needs O(n) time [140]. After the routing, the record “(i,j,pre(i),pre(j), w(i,j), PE0,j)” is sent to PE0,j, where the edge (i,j) is stored in the register T2[0,j] := (i,j), pre(i) and pre(j) are stored in the registers PT2[0,j] and PT2[0,j], respectively, and the weight of (i,j) is stored in the register W2[0,j] := w(i,j). As a result, the edges of T2 along with the pre-order numbering of their endpoints in Tin are stored in row 0.

Step 8. Broadcast each record of T2 formed in Step 7 in row 0 to the corresponding column by sending the corresponding items to the corresponding registers of the processors in that column. This can be done in O(n) time obviously. Now the tree T2 is also stored in every row.

Step 9. Find the single most vital edge, using Equation 6.3.

Step 9.1. Each PEi,j forms a routing record “(i,j,pre(i),pre(j), w(i,j), PE0,j)” if (i,j) is an edge of Tin and i is the parent of j; otherwise forms a routing record “(i,j,pre(i),pre(j), w(i,j), ∞)”. Route all records to their destinations. Then, the record “(i,j,pre(i),pre(j), w(i,j), PE0,j)” is sent to PE0,j, where the edge (i,j) is stored in the register T1[i,0] := (i,j), pre(i) and pre(j) are stored in the registers PR1[i,0] and PR2[i,0], and the weight of (i,j) is stored in the register W1[i,0]. The routing costs O(n) time. As a result, all edges of Tin are stored in column 0.

Step 9.2. Broadcast PR1[i,0] and PR2[i,0] to row i for all i, 0 ≤ i ≤ n – 1.

Step 9.3. Each PEi,j runs the following instruction: Temp[i,j] := 1 if Equation 6.5 holds; Temp[i,j] := 0 otherwise. This step can be done in O(1) time because all data needed are available in PEi,j.

Step 9.4. First PEi,0 sets the register W[i,0] with the initial value as ∞ for all i, 0 ≤ i ≤ n – 1. Then all processors in row i find an edge of T2 which has the minimum weight. This can be done by prefix computation in row i. Finally the weight of the edge found is sent to the register W[i,0] of PEi,0 which takes O(n) time.

Step 9.5. Each PEi,0 sets W[i,0] := W[i,0] – W1[i,0]. Then all processors in column 0 find the maximum value from all W[i,0], 0 ≤ i ≤ n – 1, which is not ∞. The corresponding edge in Tin is the single most vital edge by Equation 6.3.

Therefore we have the following theorem.
**Theorem 7.1** Let \( G(V,E,w) \) be an undirected, weighted graph, finding the single most vital edge w.r.t. MSTs needs \( O(n) \) time on an \( n \times n \) mesh array.

### 7.3.2 Finding the single most vital edge w.r.t. SPs

In this subsection we only deal with an undirected, weighted graph \( G(V,E,w) \). Consider the single most vital edge problem w.r.t. a shortest path between \( s \) and \( t \).

Before we proceed, let us recall the sequential algorithm given by Malik et al. [130] for this problem. Let \( T_s \) be a shortest path tree starting at \( s \), \( \pi_{s,t} \) the shortest path from \( s \) to \( t \) in \( T_s \). The tree \( T_t \) can be defined similarly. Let \( d_s(i) \) and \( d_t(i) \) be the lengths of the shortest paths from \( s \) to \( i \) in \( T_s \) and from \( t \) to \( i \) in \( T_t \) respectively, where \( i \in V \). If an edge \((i,j)\) in \( \pi_{s,t}\) is removed, the vertex set \( V \) of \( T_s \) is divided into two disjoint subsets \( W \) and \( V \setminus W \) which contain vertices \( s \) and \( t \) respectively. Define

\[
Q(i,j) = \{ (x,y) \mid (x,y) \in E, \ x \in W, \text{ and } y \in V \setminus W \} \tag{7.2}
\]

Then, their algorithm is based on the following two important observations.

**Observation 1** [130]. An edge \((i,j)\) is in \( \pi_{s,t} \) if and only if \( d_s(t) = d_t(s) = d_s(i) + w(i,j) + d_t(j) = \min_{(x,y) \in E} \{ d_s(x) + w(x,y) + d_t(y) \} \).

**Observation 2** [16]. If some edge \((i,j)\) in \( \pi_{s,t} \) is removed from \( T_s \), dividing the vertex set \( V \) into \( V_s \) and \( V \setminus V_s \) such that \( s \in V_s \) and \( t \in V \setminus V_s \), then there exists shortest paths from all other vertices in \( V \setminus V_s \) to \( t \) that do not use the edge \((i,j)\).

Note that the original Observation 2 in [130] is not correct, as shown by Bar-Noy et al. [16] by giving a counter-example.

If the single most vital edge is unique, it must be in \( \pi_{s,t} \), otherwise at least one of them is in \( \pi_{s,t} \) [124]. In our algorithm, we are interested in finding the one in \( \pi_{s,t} \). Consider the effect of removal of an edge \((i,j)\) in \( \pi_{s,t} \) on the length of the shortest path from \( s \) to \( t \). Since \( Q(i,j) \) is a cut between \( s \) and \( t \), then any shortest path from \( s \) to \( t \) in graph \( G' = (V,E - \{(i,j)\},w) \) must have an edge in \( Q(i,j) \). By Observations 1 and 2, such an edge and the length of the shortest path from \( s \) to \( t \) in \( G' \) can be computed by the following formula.

\[
\min_{(x,y) \in Q(i,j)} \{ d_s(x) + w((x,y)) + d_t(y) \} \tag{7.3}
\]

Therefore, if both \( T_s \) and \( T_t \) are given the single most vital edge can be found in time \( O(m|\pi_{s,t}|) \) by checking every edge in \( \pi_{s,t} \) using Formula 7.3. Through exploring the special property of undirected graphs, Malik et al. [130] suggested an \( O(m + n \log n) \) time algorithm for this problem using Fibonacci heap technique due to Fredman et al [59]. However their algorithm is highly sequential, a direct implementation of their algorithm on this model seems \( O(n^2) \) time needed.

In the following we first propose an inefficient sequential algorithm for the problem, then show that the algorithm can be easily implemented in \( O(n) \) time on the 2-D mesh array. The correctness of the algorithm is also based on the above two observations.

We proceed by assigning the edges of \( G \) with a new weighted function \( w_1 \) which is defined as follows. If an edge \( e \in E(T_s) \), set \( w_1(e) := +\infty \). Otherwise let \( e = (u,v) \).
We distinguish this case by the following three subcases: (i) if \( t \) is a descendant of \( v \), 
or \( v \) is a descendant of \( t \) in \( T_s \) but \( u \) does not, then 
\[ w_1(e) := d_t(v) + d_s(u) + w(u, v); \]
(ii) if \( t \) is a descendant of \( u \), or \( u \) is a descendant of \( t \) in \( T_s \) but \( v \) does not, then 
\[ w_1(e) := d_t(u) + d_s(v) + w(u, v); \]
(iii) if \( t \) is neither a descendant of \( u \) nor a descendant of \( v \) in \( T_s \), 
either \( u \) nor \( v \) is the descendant of \( t \) in \( T_s \), or both \( u \) and \( v \) are the 
descendants of \( t \) in \( T_s \), then \( w_1(e) := +\infty \).

Given two vertices \( u \) and \( v \), deciding whether they have ancestor-descendant 
relationship in \( T_s \) can be done in constant time, which is described as follows. We first 
traverse \( T_s \) by assigning pre-order numbering \( \text{pre}(v) \) and the number of descendants 
\( \text{rd}(v) \) for each vertex \( v \) in \( T_s \). We then test whether \( \text{pre}(u) \leq \text{pre}(v) < \text{pre}(u) + \text{rd}(u) \) 
holds, if it does, this indicates that \( v \) is a descendant of \( u \), and \( u \) is an ancestor of \( v \) in 
\( T_s \); otherwise they have not any ancestor-descendant relationship.

Having finishing the assignment of function \( w_1 \), we now compute the MST of 
\( G(V, E, w_1) \) with this new weighted function \( w_1 \). Let \( T_2 \) be the MST of this graph. 
Consider the tree \( T_2 \), again, define the replacement edge \( r(e) \) for each edge \( e \in T_2 \) as follows. 
Delete the edge \( e \) from \( T_2 \). As a result, the vertex set \( V \) of \( T_s \) is divided into two 
subsets \( V_s \) and \( V_t \) which contain \( s \) and \( t \) respectively. Let \( e' \in (V_s \times V_t) \cap E \) be an edge 
such that 
\[ w_1(e') = \min \{ w_1(e'') \mid e'' \in (V_s \times V_t) \cap E - \{ e \} \} \]. 
Then we define \( r(e) = e' \).
In case there does not exist any edge between \( V_s \) and \( V_t \). This means that the edge \( e \) 
is a bridge of \( G \). Therefore, it is the single most vital edge of \( G \). For convenience, we 
initially set \( r(e) = +\infty \) for every \( e \in \pi_{s,t} \).

**Lemma 7.6** Let \( e \in \pi_{s,t} \) and \( r(e) \) be defined as above. Then \( \{ r(e) \mid e \in \pi_{s,t} \} \subseteq E(T_2) \).

**Proof.** Trivial. \( \square \)

**Lemma 7.7** An edge \( e^* \in \pi_{s,t} \) is the single most vital edge of \( G(V, E, w) \) if and only if 
\( w_1(r(e^*)) = \max \{ w_1(r(e)) \mid e \in \pi_{s,t} \} \).

**Proof.** Let \( e^* = (u, v) \) and \( u \) be the immediate predecessor of \( v \) in \( \pi_{s,t} \). Assume 
that \( r(e^*) = (u', v') \). By the definition \( w(r(e^*)) = d_s(u') + d_t(v') + w(u', v') \), and 
\( w(r(e^*)) = \max \{ w(r(e)) \mid e \in \pi_{s,t} \} \). By Observations 1 and 2, the lemma follows. \( \square \)

Now we present an implementation of the proposed algorithm on the 2-D mesh 
array. From the above discussion, we see that all other operations are similar to those 
used for the single most vital edge problem \( w.r.t. \) MSTs in Section 7.3. We omit them 
here. However, the construction of \( T_s \) is different from that for the MST \( T \) of \( G(V, E, w) \) 
in Section 7.3.1. In the following we show how to construct \( T_s \). The construction of \( T_t \) is 
similar and is omitted. We proceed as follows.

**Step 1.** Compute all pairs shortest paths of \( G \). Denote by \( S \) the resulting distance 
matrix. Then, \( d_s(i) = S[i, i] \) and \( d_t(i) = S[t, i] \).

**Step 2.** Broadcast \( d_s(i) \) to row \( i \) and column \( i \), i.e., \( D_s^{[1]}[i, j] := d_s(i) \) and 
\( D_s^{[2]}[j, i] := d_s(i) \) for all \( j, 0 \leq j \leq n - 1 \).
§7.4 Algorithms on the Hypercube Array

Step 3. Construct a directed tree $T_i^m$ rooted at $s$ — a directed version of $T_i$, in which the direction of each edge is pointed to the root, described as follows. Each $PE_{i,j}$ sets its mask "1" if $d_s(i) + w(i,j) = d_s(j)$; otherwise sets its mask "0". All processors in row $i$ select a $j_0$ such that $j_0 = \min\{j \mid PE_{i,j} \text{ is masked by } "1"\}$ by prefix computation. Assume that $j_0$ has been found. Then $PE_{i,j_0}$ sets $F(j_0) := i$ and $T_s^m[i,j_0] := 1$. Meanwhile, each processor forms a record "(the source address, the destination address, label)" is also formed in for later use, where if label = 1, it indicates that this is an edge of $T_s$, otherwise it indicates that this is not an edge of $T_s$.

After Step 3, $T_s^m$ has been established. Now consider the construction of the directed tree $T_s^{out}$ in which the direction of each edge leaves from the root. It can be easily implemented by routing the records formed in Step 3. $PE_{i,j}$ sets $T_s^{out}[j,i] := 1$ if it gets a message from $PE_{j,i}$ with label = 1; otherwise sets $T_s^{out}[j,i] := 0$. Note that $T_s^m$ and $T_s^{out}$ are the corresponding version of $T_m$ and $T_{out}$ in Section 7.3.1.

Compute $pre(v)$ and $nd(v)$ for each vertex $v \in V$ by pre-order traversal on $T_s^m$, broadcast $pre(v)$ and $nd(v)$ along row $v$ and column $v$, and broadcast $pre(t)$ and $nd(t)$ to all processors on the 2-D mesh array. All these operations can be done in $O(n)$ time by the discussion in Section 7.3.1.

Next consider assigning the new weighted function $w_1$ to the edges of $G(V,E)$. Having $T_s^m$ and $T_t^m$, where $T_t^m$ is a directed version of $T_t$ rooted at $t$ and the direction of all edges in $T_t$ are pointed to the root, broadcast $d_s(v)$ and $d_t(v)$ along row $v$ and column $v$ respectively. Now for an edge $e = (i,j)$, if it is an edge of $T_t^m$, which can be checked easily, then assign $w_1(i,j) := w_1(j,i) := +\infty$. Otherwise, check whether $t$ is a descendant of $i$ in $T_s^m$, this can be done by checking whether $pre(i) \leq pre(t) < pre(i) + nd(i)$ holds. If it does, $t$ is a descendant of $i$. Otherwise $t$ is not a descendant of $i$. To check whether $j$ is a descendant of $t$ can be done similarly. The testing can be finished in $O(1)$ time because all data needed have already been stored in $PE_{i,j}$ for all $i$ and $j$, $0 \leq i,j \leq n-1$.

By the result in the initial part of this subsection, we know that the remaining part is to find the replacement edges in graph $G(V,E,w_1)$. The operations are similar to the one used in Section 7.3.1, and are omitted. Therefore we have

**Theorem 7.2** Given an undirected simple graph $G(V,E,w)$ and two specified vertices $s$ and $t$, the single most vital edge w.r.t. a shortest path between $s$ and $t$ can be found in $O(n)$ time on an $n \times n$ mesh array.

### 7.4 Algorithms on the Hypercube Array

Let us recall the hypercube array defined in Section 7.2. Each processor in this array is indexed in the form $(i,j,k)$, where $0 \leq i,j,k < n$. If we fix one parameter of the three parameters $i$, $j$ and $k$, for example, $i = 0$. Then all processors $PE_{0,j,k}$ and their interconnection structure, $0 \leq j,k < n$, form a sub-hypercube $Q$, and $Q$ may be regarded as a 2-D array in which the indexing scheme is the same as that for the 2-D mesh array but the interconnection structure among processors is different.
From the algorithms in Section 7.3, we know that they make use of the transitive closure computation which serves as a key subroutine. Dekel et al. [38] have shown how to implement this computation in $O(\log^2 n)$ time on the hypercube array by showing that matrix multiplication $C = A \times B$ can be done in $O(\log n)$ time on this model. Assume that the data distribution of $A$, $B$ and $C$ in the hypercube array is as follows:

$$A[0, j, k] := a_{j,k};$$
$$B[0, j, k] := b_{j,k}.$$

$a_{j,k}$ and $b_{j,k}$ are the elements of the two matrices to be multiplied, $0 \leq j, k < n$. $C$ is the final resulting matrix where $C[0, j, k] := c_{j,k} = \sum_{i=0}^{n-1} a_{j,i} b_{i,k}$, $0 \leq j, k < n$.

Now let $A$ be the adjacency matrix of $G$ and $A^*$ be the transitive closure of $G$. Then, it is well known that $A^* = A + A^2 + \ldots + A^n$. Suppose that $A$ and $A^*$ are stored in $Q$, i.e., an entry $a_{i,j}$ of $A$ and an entry $a^*_{i,j}$ of $A^*$ are stored in the registers $A[0, i, j]$ and $A^*[0, i, j]$ of $PE_{0,i,j}$ respectively, $0 \leq i, j < n$. It easily derives the following lemma.

**Lemma 7.8** [38] Let the adjacency matrix $A$ of $G$ is stored initially on $Q$ – a sub-hypercube in the hypercube array, the transitive closure $A^*$ of $G$ can be computed, and be finally stored on $Q$ in $O(\log^2 n)$ time on the hypercube array.

Dekel et al. also pointed out that all pairs shortest paths of $G$ can be computed, and finally stored in $Q$ in $O(\log^2 n)$ time.

Note that only the transitive closure computation and the all pairs shortest paths computation are involved in all processors of the hypercube array but all the other steps of the algorithms in Section 7.3 can be simulated on $Q$ itself in $O(\log^2 n)$ time.

We now discuss the implementation details of the data movements on this model. As said before, we may think $Q$ as a 2-D array in which the processors are hypercube-connected, all processors $PE_{0,i,j}$ with $0 \leq j < n$ form row $i$, and all processors $PE_{0,k,j}$ form column $j$ for all $k$ in $0 \leq k < n$.

Both Horizontal Rotation and Vertical Rotation operations used for computing the transitive closure and all pairs shortest paths have been implemented with $O(\log^2 n)$ time using $n^3$ hypercube-connected processors in [38]. In the following we discuss the other data movements on $Q$.

1. **Random Access Write** [140]. In this formulation, an index $(0, i', j')$ is contained in $PE_{0,i,j}$. The data from a specified register of $PE_{0,i,j}$ is to be transmitted to $PE_{0,i',j'}$, $0 \leq i, i', j, j' \leq n - 1$. It is assumed that no two processors are sending their data to the same destination, and if the destination address is $\infty$, then the data from $PE_{0,i,j}$ is not transmitted to any other processor. The details of how this data movement is implemented in $O(\log^2 n)$ time on $Q$ can be seen in [140].

2. **Horizontal Broadcasting**. Assume that in each row $i$, an element $a_i$ is stored in the register $X$ of some processor, the objective to broadcast $a_i$ to the register $X$ of each processor in row $i$ for all $i$, $0 \leq i \leq n - 1$. The procedure consists of several rounds. Initially, each processor $PE_{0,i,j}$ either forms a routing record “$(PE_{0,i,j}, PE_{0,i,0})$” if it holds $a_i$, or forms a routing record “$(PE_{0,i,j}, \infty)$” otherwise, where $PE_{0,i,0}$ is the destination address. Then we apply Random Access Write to route the records to their destinations. As a result, $a_i$ is stored in $X$ of $PE_{0,i,0}$. Third, broadcast $a_i$ to all
processors in row $i$, which is implemented as follows. Suppose that $i = r_{2q-1}r_{2q-2} \cdots r_q$. Then

\begin{verbatim}
for l := 1 to 2q − 1 do
  if $r_l = 0$ then
    copy the content of $X$ of $PE_{0,i,j}$ to the $X$ of $PE_{0,i,j}$,
    where * is either “0” or “1”;
  endif
endfor
\end{verbatim}

Finally the element in some processor in row $i$ is broadcast to all the other processors of that row. The time used is $O(\log^2 n)$ because the routing needs $O(\log^2 n)$ time.

3. Vertical Broadcasting. This is the vertical analog of Horizontal Broadcasting, and is omitted.

4. Horizontal Prefix. In this formulation, each processor $PE_{0,i,j}$ contains an element $d_{0,i,j}$ which is stored in the register $D$. Then the elements in row $i$ form a sequence $d_0, d_1, \ldots, d_{n-1}$. The problem is to compute all prefixes of this sequence, i.e., the processor $PE_{0,i,j}$ will contain the result $\sum_{k=0}^{i} d_{0,i,k}$ and will be stored in the register $S[0, i, j]$ for all $i, 0 \leq i \leq n - 1$. Note that the processors of row $i$ and their interconnection structure form a sub-hypercube. Now the problem becomes to compute all prefixes in the hypercube. In the following we present an algorithm for this propose. Our algorithm is directly derived from the algorithm for the polynomial evaluation problem on a prefect shuffle network of $n$ processors by Stone [161]. Assume that $n = 2^q$. Let $j_q-1j_{q-2} \cdots j_0$ be the binary representation of $j$. Initially $PE_{0,i,j}$ sets $S[0, i, j] := d[0, i, j]$ for all $i$ and $j$, $0 \leq i, j \leq n - 1$. Then

\begin{verbatim}
for any two neighboring processors $PE_{0,i,j}$ and $PE_{0,i,j}^{(0)}$ do
  if $j_0 = 0$ then
    $PE_{0,i,j}$ sends the content $s$ of its register $S[0, i, j]$ to $PE_{0,i,j}^{(0)}$;
    $PE_{0,i,j}^{(0)}$ sets $S[0, i, j] := S[0, i, j] + s$
  endif
endfor;
for l := 1 to $q$ − 1 do
  $PE_{0,i,j}$ sets its tag with “1” if $j_{l-1} \ldots j_0 = 1 \ldots 1$ and $j_l = 0$.
  sets $X[0, i, j] := S[0, i, j]$, a temper register for broadcasting propose;
  /* broadcast the value of $X$ of the processors with tag=1 to some neighbors */
  for p := 0 to $l$ − 1 do
    each $PE_{0,i,j}$ with tag=1 sends the content of $X$ to its neighbor $PE_{0,i,j'}$
    where $j_q-1j_{q-2} \cdots j_{l+1}' = j_q-1j_{q-2} \cdots j_{l+1}$ and $j_l = 1$;
    the receiver $PE_{0,i,j'}$ do
      if its tag is “1” then do nothing
      else copy the received content to its $X$ and set its tag by “1”;
    endif
  endfor;
  if $j_l = 1$ then
\end{verbatim}
\[ PE_{0,i,j} \text{ sets } S[0,i,j] := S[0,i,j] + X[0,i,j] \]

endif
endfor.

As a result, the content of \( S \) in \( PE_{0,i,j} \) is \( \sum_{k=0}^{j} d_{0,i,k} \). It is easy to verify the correctness of the algorithm above. The time used for computing all prefixes is \( O(\sum_{l=1}^{q-1} \sum_{p=0}^{l-1}) = O(q^2) = O(\log^2 n) \) time.

5. Vertical Prefix. This operation is similar to Horizontal Prefix and is omitted. Then, we have the following lemmas.

**Lemma 7.9** Let \( a_0, a_1, \ldots, a_{n-1} \) be stored in \( n \) different processors on \( Q \) – a sub-hypercube of the hypercube array. Along with \( a_i \), \( 0 \leq i \leq n-1 \), the destination processor address (also in \( Q \)) is also stored in the same processor. Assume that all destinations are distinct. Then routing the elements to their destinations needs \( O(\log^2 n) \) time.

**Proof.** First, \( PE_{0,i,j} \) forms a routing record \( "(PE_{0,i,j}, PE_{\text{des}(a_i)}, a_i)" \) if it contains \( a_i \); otherwise forms a routing record \( "(PE_{0,i,j}, \infty, 0)" \), where \( PE_{\text{des}(a_i)} \) is the destination of \( a_i \). Then route the records to their destinations by Random Access Write. After that, \( a_i \) is sent to \( PE_{\text{des}(a_i)} \) for all \( i, 0 \leq i \leq n-1 \). Routing takes \( O(\log^2 n) \) time. The lemma follows. \( \square \)

**Lemma 7.10** Let \( a_0, a_1, \ldots, a_{n-1} \) be stored in some row (column) \( i \) on \( Q \) – a sub-hypercube of the hypercube array, where \( a_j \) is stored in \( PE_{0,i,j} \). Broadcasting \( a_i \) to row \( i \) and column \( i \) needs \( O(\log^2 n) \) time for all \( i, 0 \leq i, j \leq n-1 \).

**Proof.** First, each processor \( PE_{0,i,j} \) forms a routing record \( "(PE_{0,i,j}, PE_{0,i,j}, a_i)" \) if \( a_i \) is stored in it, otherwise forms a routing record \( "(PE_{0,i,j}, \infty, 0)" \). Then route the records to their destinations by Random Access Write. After that, \( a_i \) is sent to the \( PE_{0,i,i} \). Finally the processors in each row (column) duplicate \( a_i \) to all processors by Horizontal Broadcasting (Vertical Broadcasting). Routing and broadcasting need \( O(\log^2 n) \) time. The lemma thus follows. \( \square \)

From the discussion above, we know that all the steps in the algorithms on the 2-D mesh array can be simulated on the hypercube array by using the above data movements operations, and each of such simulation needs \( O(\log^2 n) \) time. Therefore we have

**Theorem 7.3** Given an undirected simple graph \( G(V,E,w) \) with \( n \) vertices, the single most vital edge w.r.t. either MSTs or SPs between \( s \) and \( t \) can be found in \( O(\log^2 n) \) time using \( n \times n \times n \) processors on a hypercube-connected processor array.
7.5 Conclusions

In this chapter we presented fast algorithms for finding the single most vital edge \textit{w.r.t.} either MSTs or SPs between \( s \) and \( t \) on both the 2-D mesh array and the hypercube array. The algorithms on the 2-D mesh array are probably the first ones with the optimal time complexities. The algorithms on the hypercube array are the fastest, compared with a previously known result [83].
Chapter 8

Constructing the Spanners of Graphs in Parallel

8.1 Introduction

The sparse $t$-spanner concept has found a number of applications in distributed computing and communications network design [10, 11, 12, 13, 142, 143]. For example, Peleg and Upfal [143], Awerbuch et al. [11], and Awerbuch and Peleg [12] use $t$-spanners to design efficient routing schemes in distributed communications networks. Awerbuch [10], Peleg and Ullman [142] use them to design synchronizers. Cohen [29] uses them to solve the approximate all pairs shortest paths problem.

Regarding the spanners on some special graphs such as Euclidean graphs, geometry graphs and chordal graphs, much effort has been taken in recent several works [5, 20, 36, 37, 141, 167]. Peleg and Schäffer [141], Althöfer et al. [3] have shown some very interesting properties of $t$-spanners on unweighted and weighted graphs. Cohen [29] has suggested a randomized parallel algorithm for finding a $t$-spanner with size $O(n^{1+\frac{2}{m_t}})$ on a weighted graph. Her algorithm needs $O(\frac{W_{\max}}{n} \beta^2 \log^2 n)$ expected time with $O(n^{1/\beta} m \beta \log^2 n)$ work on an EREW PRAM, where $\beta = t / (2 + \epsilon / 2)$, $wt(e)$ is the weight of edge $e$, $W_{\max} = \max\{wt(e) \mid e \in E\}$, $W_{\min} = \min\{wt(e) \mid e \in E\}$, and $\epsilon$ is an arbitrarily small constant.

Despite the existence of several efficient sequential and distributed algorithms for finding a sparse $t$-spanner, we have not yet seen any deterministic parallel algorithm for it. In this chapter we first relax the restriction of the problem by introducing the approximate $t$-spanner concept. We then present simple parallel algorithms for finding a sparse, approximate $t$-spanner on both unweighted and weighted graphs in terms of size and weight. The algorithms exhibit some trade-offs between the running time and the factor of the spanner obtained. We finally present an NC algorithm for finding a moderate $2t$-spanner on a weighted graph $G$.

The remainder of this chapter is organized as follows. In Section 8.2 we introduce some notations and the tree decomposition concept. The algorithms for finding an approximate $t$-spanner with factors $O(t^{k+1})$ and $O(D t^{k+1})$ on both unweighted and weighted graphs are presented in Section 8.3, where $D = \max\{wt(e) \mid e \in MST\}$. The algorithm on unweighted graphs requires $O(\frac{n}{m_t} \log n)$ time and $M(n)$
processors, and the approximate \( t \)-spanner delivered has size of \( O\left( \left( \frac{n}{t^2}\right)^{1+1/t} + n \right) \), where \( M(n) \) is the number of processors needed to find a breadth-first search (BFS) tree in a graph with \( n \) vertices in time \( O(\log n) \). The algorithm on weighted graphs requires \( O\left( \left( \frac{n}{t^2}\right)^2 + \left( \frac{n}{t^2}\right)^{1+2/(t-1)} \log n \right) \) time and \( O(n^2) \) processors, and the approximate \( t \)-spanner delivered has size of \( O\left( \left( \frac{n}{t^2}\right)^{1+2/(t-1)} + n \right) \), weight of \( \left( \frac{n}{t^2}\right)^{2+\epsilon} + 1\) \( \text{wt}(\text{MST}) \).

In Section 8.4 we suggest an NC algorithm for constructing a moderate \( 2t \)-spanner on weighted graphs with a size of \( O(\min\{m, n^2, \log_1(n)\min(\frac{\text{wt}(\text{MST})}{\text{min}})\}) \). The algorithm requires \( O(\log^3 n \log_1 + \epsilon(\frac{\text{wt}(\text{MST})}{\text{min}})) \) time and \( O(n^3) \) processors, where \( \epsilon \) is a small constant with \( 0 < \epsilon < 1/2 \). We conclude our discussions in Section 8.5. Unless otherwise specified, all the proposed parallel algorithms in this chapter run on an ARBITRARY CRCW PRAM.

8.2 Preliminaries

Let \( \mathcal{X}_i (\subseteq V) \) be a vertex set such that \( \bigcup_{i=1}^{s} \mathcal{X}_i = V, 1 \leq i \leq s \leq n \). The set \( \mathcal{H} = \{ \mathcal{X}_i \mid 1 \leq i \leq s \} \) is called the coarse vertex cover on \( V \) if there exist \( X_i \) and \( X_j \) such that \( X_i \cap X_j \neq \emptyset, i \neq j \); otherwise \( \mathcal{H} \) is called the exact vertex cover on \( V \). If \( \mathcal{H} \) is an exact vertex cover on \( V \), then \( \mathcal{X} \in \mathcal{H} \) is called a vertex cluster; otherwise \( \mathcal{X} \) is simply called a partial cover. For a vertex \( u \), \( \mathcal{X} \) is called the home cover of \( u \) if \( u \in \mathcal{X} \). Obviously, every vertex \( u \in V \) has only a home cover when \( \mathcal{H} \) is an exact vertex cover on \( V \).

In order to make our algorithmic description easier later, we introduce a restricted tree decomposition concept. Let \( T(V, E_T) \) be an inverted tree with \( n \) vertices. The tree decomposition of \( T \) is to divide \( T \) into a forest of \( n' \) inverted subtrees, and every inverted subtree has no more than two levels (the root of a tree is the first level), where \( n' \leq \lfloor n/2 \rfloor \).

The above defined tree decomposition can be easily implemented in parallel. Suppose that the level number of every vertex of \( T \) has been given. Then, the vertex set \( V \) is divided into two disjoint subsets \( V_1 \) and \( V_2 \), where \( V_1 \) consists of all vertices with odd level numbers and \( V_2 \) consists of all vertices with even level numbers. If \( |V_1| \leq |V_2| \), then all vertices in \( V_1 \) are selected as the roots of subtrees. Let \( u \in V_1 \) be such a vertex, a vertex \( v \in V_2 \) is in the subtree rooted at \( u \) if and only if the level number of \( v \) is larger than that of \( u \) by one and \( u \) is the parent of \( v \) in \( T \). Otherwise the vertices in \( V_2 \) are selected as the roots of the subtrees. For this latter case, the root \( r \) of \( T \) is assigned to one of the subtrees whose root is a child of \( r \) in \( T \).

**Lemma 8.1** Let \( T \) be a forest in which the number of vertices is \( n \) and each tree has two vertices at least. Then applying the tree decomposition to \( T \) can be done in \( O(\log n) \) time using \( O(n) \) processors.

**Proof.** For every tree \( T \) in \( \mathcal{F} \), we first calculate the level numbers of vertices and \( |V_i| \) in \( T, i = 1, 2 \). This can be done in \( O(\log n) \) time with \( O(n) \) processors using Euler traversal technique. We then apply the tree decomposition to the trees with \( |V_i| \leq |V_2| \), which can be done in \( O(1) \) time with \( O(n) \) processors. Finally we apply the tree
decomposition to those trees with $|V_2| < |V_1|$. Therefore applying tree decomposition to $\mathcal{F}$ can be done in $O(\log n)$ time with $O(n)$ processors. □

8.3 Finding Approximate $t$-Spanners

8.3.1 Unweighted Graphs

Before we proceed, we introduce an algorithm for finding a $2t$-spanner first. Peleg and Schäffer [141] once gave the following lemma, and proved that the size of the spanner obtained is tight in big-O sense.

**Lemma 8.2** [141]. For every $n$-vertex graph $G$ and for every fixed $t \geq 1$, there exists a polynomial time constructible $(4t + 1)$-spanner with $O(n^{1+1/t})$ edges.

Actually Lemma 8.2 is derived directly from Awerbuch’s algorithm of constructing an optimal $\gamma$ synchronizer [10]. The basic idea of his algorithm is described as follows. The vertex set $V$ is partitioned into maximum subsets of vertices called clusters such that every cluster is connected, and the diameter of a cluster does not exceed the logarithm of its cardinality. This guarantees that the total number of neighboring cluster pairs is linear, and the maximum cluster diameter thus is logarithmic in the number of vertices of the graph.

Define the remaining graph as a subgraph induced on the vertices that were not yet joined to any cluster. The basic phase of Awerbuch’s algorithm is that a vertex in the remaining graph is chosen as the new cluster center, and a cluster centered at the vertex is then formed. This phase is repeated until the remaining graph contains no vertex. We now show that a $2t$-spanner with a size of $O(n^{1+1/t})$ can be achieved by Awerbuch’s algorithm. Let $p = n^{1/t}$. We first select an arbitrary vertex $v$ as the root of a BFS tree. We then generate the BFS tree level by level until it reaches some level $l_0$ such that the number of vertices in level $l_0$ is less than $p - 1$ times the total number of vertices in all previous levels. The level $l_0$ is the rejected level. The procedure of generating clusters is terminated at level $l_0$, and all vertices with level number $l \leq l_0$ are included in cluster $C_v$ centered at $v$. When a cluster $C$ is formed, its rejected level is examined. If the rejected level is not empty, a vertex $u$ in it is then chosen as the new cluster center. In case the rejected level of cluster $C$ is empty, the algorithm backtracks to the cluster from which $C$ itself is found recursively. The resulting spanner $G' = (V, E_1 \cup E_2)$ of $G$ is expressed as follows:

$$E_1 = \cup_{all \; clusters \; \mathcal{C}} \{e \mid e \text{ is an edge of the BFS tree in } \mathcal{C}\}$$

and

$$E_2 = \{(u, v) \mid (u, v) \in E, u \in \mathcal{C}_u \text{ and } v \in \mathcal{C}_v\}.$$

Thus, the size of the spanner $G'$ is $|E_1| + |E_2| \leq (n - 1) + pn = O(n^{1+1/t})$, and the factor of $G'$ is $2t$ which will be shown later.

The parallelization of Awerbuch’s algorithm is not very difficult. One such a parallel version is given as follows.
Algorithm 1

Step 1. Initialization.

\[ A(i) := 0; B(i) := 0; count := 0; \]

// \( A(i) = 0 \) that means vertex \( i \) does not yet included in any clusters.*/

// \( B(i) \) means that vertex \( i \) is explored by the \( B(i) \)th cluster */

// \( count \) is the number of clusters in \( G \) */

Step 2. \textbf{while} there is a vertex \( i \) with \( A(i) = 0 \) \textbf{do}

2.1. \( \text{count} := \text{count} + 1; \)

\[ \text{select a vertex } i \text{ as the root of the new BFS tree such that} \]

\[ A(i) = 0 \text{ and } B(i) = \max\{B(j) \mid j = 1, \ldots, n\}; \]

\[ A(i) := 1; B(i) := \text{count}; \]

2.2. \( \text{generate a BFS tree } T_i \text{ rooted at } i \text{ such that the diameter of } T_i \text{ doesn't exceed the logarithm of its cardinality}; \)

2.3. \( \text{for each vertex } v \text{ in } T_i, A(v) := 1; \)

2.4. \( \text{for each vertex } v \text{ in the rejected level of } T_i, B(v) := \text{count}; \)

\textbf{endwhile.}

Now we have the following lemma.

Lemma 8.3 \textit{Let} \( G(V,E) \) \textit{be an unweighted, connected graph. Then a} \( 2t \)-spanner of \( G \) \textit{with a size of} \( O(n^{1+1/4}) \) \textit{can be obtained in} \( O(n \log n) \) \textit{time with} \( M(n) \) \textit{processors.}

\textbf{Proof.} By Algorithm 1, the initialization can be done in \( O(1) \) time using \( O(n) \) processors. Generating a cluster and labeling the cluster can be done in \( O(\log n) \) time with \( M(n) \) processors. Note that \( M(n) \geq \Omega(n^2) \), currently the best result of \( M(n) = n^{2.376} \) [32]. Selecting the center for a new cluster needs \( O(\log n) \) time and \( O(n/\log n) \) processors by prefix computation. The number of iterations of the \textbf{while} loop is at most \( O(n) \). Obviously the total number of edges in all BFS trees is at most \( n - 1 \). By the definition of clusters, there are at most \( pn' \) edges between a cluster and all the other clusters if the cluster has \( n' \) vertices. Therefore the resulting spanner has \( O(pn + n - 1) = O(n^{1+1/4}) \) edges.

We then consider the factor of the spanner \( G' \). Assume that an edge \( (u,v) \in E \) does not included in \( G' \), if both \( u \) and \( v \) are in a cluster \( C_w \) centered at \( w \), then the distance between \( u \) and \( v \) in \( G' \) is at most the distance between \( u \) and \( w \) in \( G' \) plus the distance between \( v \) and \( w \) in \( G' \). So, the distance between \( u \) and \( v \) in \( G' \) is no more than \( 2 \log p n = 2t \). If \( u \in C_u \) and \( v \in C_v \), then the distance between \( u \) and \( v \) in \( G' \) is one because \( (u,v) \in G' \) by the algorithm. \( \square \)

Having Algorithm 1, we now show how to find an approximate \( t \)-spanner on \( G \) in parallel. Our algorithm consists of several phases. In each phase we compress those vertices whose distances are not far away from each other into a supervertex (also called a cluster). Then, a supergraph \( \mathcal{G}(V,E) \) is formed, where \( V \) consists of all supervertices and there is an edge in \( E \) if and only if there is at least an edge of \( G \) between the two supervertices (we keep track of the corresponding edge of \( G \)). So, the approximate spanner, denoted by \( SP(G) \), of \( G \) can be expressed as \( SP(\mathcal{G}) \cup \)
\{the tree edges in clusters\} recursively. The detailed algorithm for finding \(SP(G)\) is given below, where the function \(F\) on \(\mathcal{V}\) defines a forest of inverted trees and \(F(v)\) is the parent of \(v\).

Algorithm 2
\[
i := 1; \mathcal{V} := \mathcal{V};
\]
\[
\text{while } i \leq k \text{ do}
\]
\[
\text{for } j := 1 \text{ to } \lfloor \log_3 t \rfloor \text{ do}
\]
\[
\text{Step 1. for each vertex } v \text{ in } \mathcal{V}, \text{ set } F(v) := v;
\]
\[
\text{Step 2. for each vertex } v, \text{ find a neighbor } u \text{ with the smallest index, and set } F(v) := u;
\]
\[
\text{if } F(F(v)) = v \text{ and } F(v) \neq v \text{ then }
\]
\[
\text{if } F(v) > v \text{ then } F(v) := v \text{ else } F(F(v)) := F(v) \text{ endif}
\]
\[
\text{endif;}
\]
\[
/* \text{ delete all cycles consisting of two edges, */}
\]
\[
/* \text{ As a result, we obtain a forest } \mathcal{F} \text{ of inverted trees. */}
\]
\[
/* \text{ The forest } \mathcal{F} \text{ now contains all inverted trees formed in Step 2, */}
\]
\[
/* \text{ and each inverted tree contains at least two vertices. */}
\]
\[
\text{Step 3. generate a forest } \mathcal{F}' \text{ by applying the tree decomposition to } \mathcal{F},
\]
\[
denote \text{ by } E_{i,j} \text{ the edge set of } \mathcal{F}';
\]
\[
/* \text{ Now every tree is partitioned into many subtrees. */}
\]
\[
\text{Step 4. construct a supergraph } \mathcal{G}(\mathcal{V}, \mathcal{E}) \text{ such that each subtree in } \mathcal{F}'
\]
\[
is a supervertex and there is an edge in } \mathcal{E} \text{ if there is an edge between the two supervertices;}
\]
\[
\text{endfor;}
\]
\[
i := i + 1;
\]
\[
\text{endwhile}
\]
\[
\text{Step 5. find a } 2t\text{-spanner } \mathcal{G}' \text{ on } \mathcal{G}(\mathcal{V}, \mathcal{E}) \text{ in parallel by Algorithm 1;}
\]
\[
\text{Step 6. The approximate } t\text{-spanner, } SP(G), \text{ of } G \text{ is } G(\mathcal{V} \cup \bigcup_{i=1,..,k} E_{i,j}) \cup \mathcal{G}'.
\]

**Theorem 8.1** Given an unweighted, connected graph \(G(\mathcal{V}, \mathcal{E})\), an approximate \(t\)-spanner with a factor of \(O(k^{k+1})\) can be constructed in \(O(\frac{n}{\log n})\) time using \(M(n)\) processors.

**Proof.** By the above proposed algorithm, we know that Steps 1–4 can be finished in \(O(k \log_3 t \log n)\) time using at most \(O(m + n)\) processors. Step 5 can be done in \(O(\frac{n}{\log n})\) time using \(M(n)\) processors by Lemma 8.3, and this step is also the dominant step of the proposed algorithm. \(\square\)

**Theorem 8.2** The sparse, approximate \(t\)-spanner obtained by Algorithm 2 has size of \(O((\frac{n}{\log n})^{1+1/t} + n)\) and the factor of \(O(k^{k+1})\), where \(k\) is a fixed constant, \(1 \leq k \leq c \log n\) with \(c < 1\).

**Proof.** By Lemma 8.1, we know that the number of supervertices of a supergraph is at most half of the number of vertices of its immediately precedent supergraph, assuming
that $G$ is an initial supergraph. Let the number of supervertices of current supergraph be $n_i (n_1 = n)$. Then the number of supervertices of the resulting supergraph after finishing the for loop is $O\left(\frac{n}{t}\right)$. Therefore, the final supergraph has $O\left(\frac{n}{t}\right)$ supervertices after finishing the while loop. The $2t$-spanner of the final supergraph has size of $O\left(\left(\frac{n}{t}\right)^{1+1/t}\right)$ by Lemma 8.3. Note that, $|E_{i,j}| \leq |E_{i,j-1}|/2$ with fixed $i$ and $|E_{1,1}| \leq n-1$ because $E_{i,j}$ is the edge set of a forest of inverted trees. So $\sum_{j=1}^{\log_3 t} |E_{i,j}| \leq O\left(\frac{n}{t}\right)$. Therefore, the size of the resulting approximate spanner is $O\left(\left(\frac{n}{t}\right)^{1+1/t} + n\right)$.

Now we calculate the factor of the spanner. Let $d_i$ be the diameter of a cluster in the $l$th iteration of variable $j$ for fixed $i$ in Algorithm 2. Initially every vertex of $G$ is a supervertex and the diameter of every supervertex is $d_0 (= 0)$ when $i = 0$. The equation is described as follows.

$$d_l = 3d_{l-1} + 2, \quad 1 \leq l \leq \lfloor \log_3 t \rfloor.$$ 

From this equation we derive that $d_{\lfloor \log_3 t \rfloor} = t - 1$. Therefore, the maximum distance between two adjacent vertices of $G$ in a supervertex is at most $O(t^k)$, and the factor of the resulting spanner is $O(2t\cdot (\epsilon t^k + 1)) = O(t^{k+1})$, where $\epsilon$ is a constant. □

**Corollary 8.1** Given a connected unweighted graph $G(V,E)$, a $(4t+1)$-spanner with a size of $O(\min\{m, (\frac{n}{t})^2\})$ can be obtained in $O(\log n)$ time using $O(m+n)$ processors.

**Proof.** By setting $k = 1$ and running the steps from 1 to 4 of Algorithm 2, it gives a supergraph $G(V,E)$. Having the supergraph, a $(4t+1)$-spanner $G(V,E_1 \cup E_2)$ of $G$ is constructed as follows. $E_1 = \cup_{j=1}^{\log_3 t} E_{1,j}$, and an edge $(x_0,y_0)$ is added to $E_2$ if there is at least an edge $(x,y) \in E, x \in v_1, \text{ and } y \in v_2$, where $v_i \in V$ for $i = 1,2$. The size of the spanner is $|E_1| + |E_2| \leq 2n + \left(\epsilon \frac{n}{t}\right)^2 = O(\min\{m, (\frac{n}{t})^2\})$, and the factor of the spanner is $4t + 1$, which is shown as follows. Consider an edge $(x,y) \in E$ but $(x,y) \not\in E_1 \cup E_2$. If $x, y \in V$, then the distance between them in the spanner is at most $2t$. Otherwise, let $x, x_0 \in v_1$ and $y, y_0 \in v_2$ and the edge between $v_1$ and $v_2$ be $(x_0, y_0)$, then the distance between $x$ and $y$ in the spanner is the summation of the distance between $x$ and $x_0$, plus the distance between $y$ and $y_0$, and plus one (the edge $(x_0, y_0)$) which equals at most $4t + 1$. □

### 8.3.2 Weighted Graphs

Althöfer et al. [3] first consider the problem of finding a sparse $t$-spanner $G'(V, E')$ on a non-negative weighted graph $G(V, E)$ by presenting a simple greedy algorithm for it. Let $d_G(x, y)$ be the length of the shortest path between vertices $x$ and $y$ in $G$. Then their algorithm is described as follows.

**Algorithm 3**

Step 1. Initialization
1.1. sort $E$ by non-decreasing weight;
1.2. $G' := (V, \emptyset);$

...
Step 2. for every edge $e = (u, v)$ in $E$ (from the sorted list) do

2.1. compute $P(u, v)$, the shortest path between $u$ and $v$ in $G'$;

2.2. if $d_G(u, v) > t \times d_G(u, v)$ then

\[ E' := E' \cup \{(u, v)\}; \ G' := (V, E') \] endif;

endfor

Step 3. output $G'$.

If the fastest algorithm for finding a shortest path between two vertices [59] is employed, Algorithm 3 can be implemented in time $O(mn + mn \log n) = O(n^4)$. The size of $G'$ is $O(n^{1+2/(t-1)})$ and the weight of $G'$ is less than $(\frac{n}{t-1} + 1)wt(MST)$ [3]. By a considerably improved analysis of this algorithm, Chandra et al. [20] have shown that the running time of Algorithm 3 is $O(n^{3+4/(t-1)})$, and the weight of $G'$ is no more than $O(n^{2+4/(t-1)}wt(MST))$, where $\epsilon > 0$ is any arbitrarily small constant.

The naive parallel version of Algorithm 3 requires $O(m + Rn \log n)$ time if $O(n^2)$ processors are available, where $R$ is the size of $G'$ because finding a shortest path between two vertices requires $O(n \log n)$ time using $O(n^2)$ processors. However, we observe that $G'$ is such an augmented graph that each time when a new edge is added to it, the shortest path between the endpoints of the edge on the augmented graph is computed. This is a partially dynamic maintenance problem. Therefore, the partially dynamic parallel algorithm for the maintenance of all pairs shortest paths, developed by Liang et al. [120] (also see Chapter 3), can be used. Their algorithm requires $O(\log n)$ time using $O(n^2)$ processors per update. We thus obtain the following lemma.

**Lemma 8.4** Given a weighted, connected graph $G(V, E)$, finding a sparse t-spanner of $G$ can be done in $O(m + n^{1+2/(t-1)} \log n)$ time using $O(n^2)$ processors. The size and the weight of the spanner obtained are $O(n^{1+2/(t-1)})$ and $O(n^{2+4/(t-1)}wt(MST))$ respectively, where $\epsilon (> 0)$ is any arbitrarily small constant.

**Proof.** Initially we compute the distance matrix of $G$. It can be easily done in $O(n \log n)$ time using $O(n^2)$ processors. Then we construct $n$ single source shortest path trees for $G'$. When a new edge is inserted to $G'$, we update the distance matrix of $G'$ which can be done in $O(1)$ time using $O(n^2)$ processors, and maintain the data structures for the $n$ single source shortest path trees which requires $O(\log n)$ time and $O(n^2)$ processors. There are at most $R$ edges to be inserted to $G'$ and $R = O(n^{1+2/(t-1)})$ [3]. Deciding whether or not an edge of $G$ is in $G'$ can be done in $O(1)$ time by checking the corresponding entries of the distance matrices of $G$ and $G'$. The algorithm therefore requires $O(m + n^{1+2/(t-1)} \log n)$ time provided $O(n^2)$ processors are available. □

We now present a parallel algorithm for finding an approximate t-spanner on weighted graphs. The idea for this case is similar to the one used for unweighted graphs. However, we must make some minor changes in order to extend our algorithm on unweighted graphs to weighted graphs. That is, we concentrate on the construction of the supergraph in each phase. Instead of finding an arbitrary edge between two supervertices served as an edge in the resulting supergraph, an edge of $G$ between
two supervertices is selected as the edge in the supergraph when it has the minimum weight, compared with the weight of all the other edges between the two supervertices. The algorithm is the same as Algorithm 2 except the following steps.

Algorithm 4

\begin{itemize}
  \item \textbf{Step 2.} for each vertex \( v \), find a neighbor \( u \) with the minimum weight compared with the other neighbors; \( F(v) := u; \)
  \item if \( F(F(v)) = v \) and \( F(v) \neq v \) then
    \item if \( F(v) > v \) then \( F(v) := v \) else \( F(F(v)) := F(v) \) endif
  \end{itemize}

\textit{/* As a result, we obtain a forest \( F \) of inverted trees. */}

\begin{itemize}
  \item \textbf{Step 4.} construct a weighted supergraph \( G(V,E) \) such that each subtree is a supervertex and an edge \((u,v)\) is selected as the edge of \( E \) if (i) \( u \in C_u \) and \( v \in C_v \); (ii) the weight of \((u,v)\) is minimum, compared with the weights of the other edges between \( C_u \) and \( C_v \);
  \item \textbf{Step 5.} find a \( t \)-spanner \( G' \) of \( G(V,E) \);
\end{itemize}

Now we analyze the computational complexity of the proposed algorithm.

**Theorem 8.3** Given a connected, weighted graph \( G(V,E) \), let \( D \) be the maximum edge weight of the MST of \( G \). Then finding an approximate, sparse \( t \)-spanner with a factor of \( O(Dt^{k+1}) \) requires \( O((\frac{n}{D})^2 + (\frac{n}{D})^{1+2/(t-1)} \log n) \) time and \( O(n^2) \) processors.

**Proof.** By Algorithm 4, Steps from 1 to 4 can be done in \( O(k \log_3 t \log n) \) time using \( O(m + n) \) processors. Step 5 can be done in \( O((\frac{n}{D})^2 + (\frac{n}{D})^{1+2/(t-1)} \log n) \) time using \( O(n^2) \) processors by Lemma 8.4. \( \square \)

**Theorem 8.4** Let \( SP(G) \) be the approximate \( t \)-spanner of \( G \) obtained by Algorithm 4 for fixed \( k \). Then, \( SP(G) \) has size of \( O((\frac{n}{D})^{1+2/(t-1)} + n) \), weight of \( O(((\frac{n}{D})^{2/1-t} + 1)wt(MST)) \), and factor of \( O(Dt^{k+1}) \), where \( \epsilon > 0 \) is any arbitrarily small constant.

**Proof.** Based on the same argument as in Section 8.3.1, we know that the number of supervertices in a weighted supergraph is at most half of the number of its immediately precedent supergraph, assuming that \( G \) is the initial supergraph. Let the current supergraph have \( n_i \) vertices \( (n_1 = n) \). Then the number of supervertices of the resulting supergraph after finishing the for loop is at most \( O(\frac{n}{D}) \). We therefore obtain the final supergraph with \( O(\frac{n}{D}) \) supervertices after finishing the while loop. Note that,
\[ |E_{i,j}| \leq |E_{i,j-1}|/2 \] with fixed \( i \) and \( |E_{1,1}| \leq n-1 \) because \( E_{i,j} \) is the edge set of a forest.

So, \( \sum_{j=1, \ldots, \lceil \log_3 t \rceil} |E_{i,j}| \leq O(\sqrt{t}) \). Note that \( \cup_{i=1, \ldots, k} E_{i,j} \subseteq \{ e : e \text{ is an edge of an MST}\} \).

A supergraph, denoted by \( G_1 \), with \( O(\sqrt{t}) \) vertices thus is obtained after finishing Step 4.

By applying the analysis in [3, 20], we know that the weight of the \( t \)-spanner of \( G_1 \) is less than \( O((\frac{t}{\sqrt{t}})^{\frac{1}{1+2/t}} wt(MST_1)) = O((\frac{t}{\sqrt{t}})^{\frac{1}{1+2/t}} wt(MST)) \) because \( wt(MST_1) \leq wt(MST) \).

The size of the spanner is \( O((\frac{t}{\sqrt{t}})^{\frac{1}{1+2/t}}) \) where \( MST_1 \) is the MST of \( G_1 \).

We analyze the factor of the spanner below. Let \( d_i \) be the diameter of a cluster obtained in the \( l \)th iteration of variable \( j \) for fixed \( i \). Initially the diameter of every supervertex is at most \( d_1 \) when \( i = 1 \), where \( 0 \leq d_1 \leq D \). The equation is expressed as follows.

\[
d_i = 3d_{i-1} + 2, \quad 2 \leq i \leq \lceil \log_3 t \rceil
\]

From this equation we derive that \( d_{\lceil \log_3 t \rceil} = \frac{t}{3}(D + 1) - 1 = O(Dt) \). Therefore the maximum distance between two adjacent vertices of \( G \) in a supervertex of \( G_1 \) is \( O(Dt^k) \). Thus, the factor of the resulting spanner of \( G \) is \( O(t(Dt^k + 1)) = O(Dt^{k+1}) \).

\[ \square \]

8.4 NC Algorithms for Finding Moderate 2\( t \)-Spanners

8.4.1 The algorithm for unweighted graphs

In Corollary 8.1 we already obtained an NC algorithm for finding a \((4t + 1)\)-spanner of unweighted graphs with a size of \( O((\frac{t}{3})^2) \). By adjusting the upper bound of \( j \) in Algorithm 2 to \( \lceil \log_3 t \rceil / 2 \), we can easily derive an NC algorithm for finding a \((2t + 1)\)-spanner with a size of \( O((\frac{t}{3})^2) \). In the following we present an NC algorithm for finding a \( 2t \)-spanner of \( G \) with size \( O(\min\{m, n^2/t\}) \) which is obviously inferior to the result in Corollary 8.1. However, the motivation behind this algorithm is to use it to devise an NC algorithm for finding a moderate \( 2t \)-spanner on weighted graphs.

Let \( d(u, v) \) be the distance between vertices \( u \) and \( v \) in \( G \). Assume that each vertex of \( G \) is on a cycle at least. Then the degree of each vertex in \( G \) is at least 2. In the following we assume that \( G \) has these two properties. If \( G \) does not have the properties, we just work on its subgraph \( H \) induced by vertices satisfying the properties. The \( t \)-spanner of \( G \) is obtained by adding the deleted vertices as well as the edges incident to them back to the resulting \( t \)-spanner of \( H \).

**Lemma 8.5** Let \( H \) be the subgraph induced on a vertex set in which each vertex is on a cycle of \( G \) at least. Then the construction of \( H \) requires \( O(\log n) \) time and \( O(m + n) \) processors on a CRCW PRAM.

**Proof.** We first construct a spanning tree \( T \) of \( G \), which requires \( O(\log n) \) time and \( O(m + n) \) processors [14]. We then label those tree edges that are covered by non-tree edges using the method in Chapters 5 and 6. This can be done in \( O(\log n) \) time using \( O(m) \) processors. We finally label the two endpoints of a tree edge by “1” if the edge
is covered; we label the endpoints of this edge with “0” otherwise. This step can be done in $O(1)$ time using $O(n)$ processors. As a result, a vertex of $G$ that does not be contained by any cycle is the vertex labeled by “0”. The remaining graph $H$ is obtained by removing all vertices labeled by “0” as well as the edges incident to them from $G$. The lemma thus follows. □

We now construct a new graph $G_i(V,E_i)$ such that an edge $(u,v) \in E_i$ if and only if $d(u,v) \leq 2^i$, $0 \leq i \leq \lceil \log n \rceil$. Denote by $U_i$ a maximal independent set of vertices on $G_i(V,E_i)$, and $T(u,i)$ a BFS tree rooted at $u$ with height $2^i$ in $G(V,E)$.

**Lemma 8.6** Given a connected undirected graph $G(V,E)$, the construction of $G_i(V,E_i)$ defined above can be done in $O(i)$ time using $O(n^3)$ processors, $0 \leq i \leq \lceil \log n \rceil$.

**Proof.** Let $G_0(V,E_0) = G(V,E)$. Assume that $G_{i-1}$ already exists. Now we construct $G_i$ as follows: for any two vertices $u$ and $v$, we add to $E_i$ the edge $(u,v)$ if $(u,v) \in E_{i-1}$, or there is at least a vertex $w$ ($w \neq u$ and $w \neq v$) such that $(u,w) \in E_{i-1}$ and $(v,w) \in E_{i-1}$. So, $G_i$ can be obtained in $O(1)$ time using $O(n^3)$ processors if $G_{i-1}$ is given. □

**Lemma 8.7** Let the graph $G_{\lceil \log t \rceil-1}(V,E_{\lceil \log t \rceil-1})$ be defined as above. Then, $d(u,v) > 2^{\lceil \log t \rceil-1} \geq t/4$ for any two non-adjacent vertices $u$ and $v$.

The proof of Lemma 8.7 is so obvious, we omit it. In the following we give an NC algorithm for constructing a $2t$-spanner of $G$ with a size of $O(min\{m, n^2/t\})$.

**Algorithm 5**

1. construct the graph $G_{\lceil \log t \rceil-1}(V,E_{\lceil \log t \rceil-1})$;
2. find a maximal independent set $U_{\lceil \log t \rceil-1}$ of $G_{\lceil \log t \rceil-1}$ in parallel;
3. for every $u \in U_{\lceil \log t \rceil-1}$, construct the tree $T(u,\lceil \log t \rceil)$;
4. The $2t$-spanner of $G$ is $\cup_{u \in U_{\lceil \log t \rceil-1}} T(u,\lceil \log t \rceil)$.

**Lemma 8.8** $\frac{n}{\Delta \lceil \log t \rceil-1+1} \leq |U_{\lceil \log t \rceil-1}| \leq \frac{4m}{t}$.

**Proof.** It is well known that, for a graph $G$ with maximum degree $\Delta$ and minimum degree $\delta$, the size of a maximal independent set $U$ of $G$ is in between $n/(\Delta + 1)$ and $n/\delta$. Then, the maximum and the minimum degrees of graph $G_{\lceil \log t \rceil-1}$ are $\Delta_{\lceil \log t \rceil-1}$ and $\delta_{\lceil \log t \rceil-1} \geq t/4$ respectively. Note that we assume that the degree of every vertex in $G$ is 2 at least. The lemma follows. □

Let $\Gamma(u,\lceil \log t \rceil)$ be a vertex set consisting of all vertices in $T(u,\lceil \log t \rceil)$ and $u$ be the center of $\Gamma(u,\lceil \log t \rceil)$. By the definition in Section 8.2, $\Gamma(u,\lceil \log t \rceil)$ is a partial cover and $\mathcal{H} = \{\Gamma(u,\lceil \log t \rceil) \mid u \in U_{\lceil \log t \rceil-1}\}$ is a coarse vertex cover on $V$.

**Lemma 8.9** Let $v \in X_i$ for all $i$, and $x_i$ be the center of $X_i$, where $X_i \in \mathcal{H}$ and $1 \leq i \leq |U_{\lceil \log t \rceil-1}|$. Then, there exists at most a partial cover $X_j$ such that $d(v,x_j) \leq 2^{\lceil \log t \rceil-2}$ but $d(v,x_k) > 2^{\lceil \log t \rceil-2}$ for all the other $X_k$ with $j \neq k$. 


Proof. If there exists two vertices \( x_j \) and \( x_k \) such that \( d(v, x_j) \leq 2^{\log t} - 2 \) and \( d(v, x_k) \leq 2^{\log t} - 2 \), then \( d(x_j, x_k) \leq d(v, x_j) + d(v, x_k) \leq 2^{\log t} - 1 \). By the construction of \( G_{\log t - 1} \), there must have an edge \( (x_j, x_k) \) connected \( x_j \) and \( x_k \), which contradicts the assumption that both \( x_j \) and \( x_k \) are included in the maximal independent set \( U_{\log t - 1} \). □

Lemma 8.10 For any \( u \in U_{\log t - 1} \), \( |\Gamma(u, \log t)| \geq t/2 \).

Proof. By the discussion in Lemma 8.8, the degree of vertex \( u \) in \( G_{\log t} \) is at least \( t/2 \). □

Lemma 8.11 Let \( \mathcal{H} \) be defined as above. Then, at least \( t|U_{\log t - 1}|/8 \) vertices has only one home cover.

Proof. We first show that the vertex set \( U_{\log t - 1} \) is an independent set of graph \( G_{\log t - 2} \). Assume that two vertices \( u \) and \( v \) in \( U_{\log t - 1} \) are adjacent in graph \( G_{\log t - 2} \); then \( d(u, v) \leq 2^{\log t} - 2 \), so, \( u \) and \( v \) are adjacent in \( G_{\log t - 1} \), which contradicts the assumption that \( U_{\log t - 1} \) is a maximal independent set of \( G_{\log t - 1} \). Recall that \( T(u, \log t - 2) \) is a BFS tree rooted at \( u \) of \( G(V, E) \) with height \( 2^{\log t - 2} \). Denote by \( \Gamma(u, \log t - 2) \) the set of all vertices in \( T(u, \log t - 2) \). Obviously, \( \Gamma(u, \log t - 2) \subseteq \Gamma(u, \log t - 1) \). We then show that \( \Gamma(u, \log t - 2) \cap \Gamma(v, \log t - 2) = \emptyset \) for any \( u \) and \( v \) \((u, v \in U_{\log t - 1} \) and \( u \neq v \)). If not, there exists at least a vertex \( w \in \Gamma(u, \log t - 2) \cap \Gamma(v, \log t - 2) \), then \( d(u, v) \leq d(v, w) + d(u, w) \leq 2^{\log t - 2} + 2^{\log t - 2} \leq 2^{\log t - 1} \). By the construction of \( G_{\log t - 1} \), \( u \) and \( v \) must be adjacent in this graph. Contradiction. It is easy to derive \( |\Gamma(v, \log t - 2)| \geq t/8 \) by the similar discussion as in Lemma 8.8, we therefore conclude that at least \( t|U_{\log t - 1}|/8 \) vertices belong to one home cover only. □

Theorem 8.5 The spanner obtained by Algorithm 5 has size of \( O(\min\{m, n^2/t\}) \) and factor of \( 2t \).

Proof. By Lemma 8.8, \( |U_{\log t - 1}| \leq 4n/t \) and a vertex \( v \in V \) belongs to at most \( 4n/t \) home covers. Therefore, the size of the resulting spanner is \( O(\min\{m, n^2/t\}) \).

The rest is to show that the factor of the spanner is \( 2t \). To achieve this bound, we only show that for every two vertices \( u \) and \( v \) such that \( d(u, v) \leq 2^{\log t - 1} \), there exists at least one home cover \( \mathcal{H} \) including the two vertices, i.e., the distance between \( u \) and \( v \) in the spanner is no more than \( 2t \). The proof proceeds as follows. By Lemma 8.7, it is impossible that both \( u \) and \( v \) are in \( U_{\log t - 1} \). Then there are four cases: (i) If either one of them is in \( U_{\log t - 1} \), say \( u \in U_{\log t - 1} \), then \( v \) is in the home cover of \( u \) because of \( d(u, v) \leq 2^{\log t} \). As a result, the distance between \( u \) and \( v \) in the spanner is no more than \( 2^{\log t} \leq t \); (ii) if neither \( u \) nor \( v \) is in \( U_{\log t - 1} \) but their common neighbor \( w \) of \( G_{\log t - 1} \) is in \( U_{\log t - 1} \), then \( w \) is the center of their home cover.
cover because \(d(v, w) \leq 2^{[\log t] - 1}\) and \(d(u, w) \leq 2^{[\log t] - 1}\), the theorem follows; (iii) if \(w\) is a neighboring vertex of \(u\) but \(v\) in \(G_{[\log t] - 1}\), then \(v\) also belongs to the home cover centered at \(w\) because \(d(v, w) \leq d(w, u) + d(u, v) \leq 2^{[\log t] - 1} + 2^{[\log t] - 1} \leq 2^{[\log t]};\) (iv) otherwise, the distance between a vertex in \(U_{[\log t] - 1}\) and \(u\) (or \(v\)) in \(G_{[\log t] - 1}\) is at least larger than 2. Then, either \(u\) or \(v\) must belong to \(U_{[\log t] - 1}\) because \(U_{[\log t] - 1}\) is a maximal independent set of \(G_{[\log t] - 1}\). This results in a contradiction. □

### 8.4.2 The algorithm for weighted graphs

Based on the *neighborhood cover* of Awerbuch and Peleg [13], Cohen [29] introduces the *pairwise cover* concept and presents an efficient sequential algorithm and a randomized parallel algorithm for finding a sparse \(t\)-spanner on weighted graphs. The basic idea of Cohen’s algorithm is to employ a logarithmic number of pairwise covers for different values of \(W\) to construct spanners. Define the *radius* of a partial cover \(\mathcal{X}\) by the distance from the center of \(\mathcal{X}\) to the farthest vertex in it. Let \(\epsilon'\) be any constant bounded by \(0 < \epsilon' < 1/2\). Cohen’s algorithm is described as follows.

**Algorithm 6**

Step 1. Initialization.

1.1. \(W_{\text{max}} := \max \{wt(e) \mid e \in E\};\)

\(W_{\text{min}} := \min \{wt(e) \mid e \in E\};\)

\(R := \lfloor \log_{1 + \epsilon'} \left( \frac{W_{\text{max}}}{W_{\text{min}}} \right) \rfloor;\)

1.2. \(\text{for } i := 0 \text{ to } R \text{ pardo} \)

\(W_i := W_{\text{min}} \left( 1 + \epsilon' \right)^i\)

\(\text{endfor;}\)

Step 2. \(\text{for } i := 0 \text{ to } R \text{ pardo} \)

construct a coarse vertex cover \(\mathcal{H}_i\) such that the radii of

every partial cover \(\mathcal{X} \in \mathcal{H}_i\) is no more than \(2^{[\log t]} W_i\)

\(\text{endfor;}\)

Step 3. The spanner obtained is \(\bigcup_{i=1}^{R} \{T_i(u, [\log t]) \mid u \text{ is the center of } \mathcal{X}, \mathcal{X} \in \mathcal{H}_i\}.\)

The correctness of the above algorithm has been justified in [29]. The key part of this algorithm is how to construct a coarse vertex cover on \(V\) (called pairwise cover in Cohen’s algorithm) efficiently in parallel. In doing so Cohen [29] introduces randomness to choose the centers of covers. However, it seems not easy to transform her randomized parallel algorithm into a deterministic version. Here we present an efficient, deterministic parallel algorithm for constructing such a coarse vertex cover by extending our technique for unweighted graphs to weighted graphs. Let \(A\) be the weighted adjacency matrix of \(G\) and \(A' = A^{-1} \odot A^{-1}\), where \(\odot\) operation function is defined as follows: \(a'_{(u,v)} = \min_{w \in V} \{a^{-1}_{(u,v)} + a^{-1}_{(w,v)}\}\) and an entry \(a'_{(u,v)}\) of \(A'\) represents the distance between \(u\) and \(v\) with length \(2^t\). Assuming \(A^0 = A\). A coarse vertex cover \(\mathcal{H}_i\) is then constructed as follows.

**Algorithm 7**

Step 1. construct an auxiliary graph \(G_i(V, E^*)\); an edge \((u, v)\) is added to \(E^*\)
if and only if \(d(u, v) \leq 2^{\log t - 1} W_i\);

Step 2. find a maximal independent set \(U(i, [\log t] - 1)\) of \(G_i(V, E^*)\) in parallel; 
Step 3. build a shortest path tree rooted at \(u\), \(T_i(u, [\log t])\) in \(G(V, E)\) such that a vertex \(v\) is included it if and only if \(d(u, v) \leq 2^{\log t} W_i\) for each \(u \in U(i, [\log t] - 1)\); 
Step 4. As a result, a coarse vertex cover on \(V\) is built as follow: 
\[
\mathcal{H}_i = \{T_i(u, [\log t]) \mid u \in U(i, [\log t] - 1)\}.
\]

**Lemma 8.12** Given a weighted graph \(G(V, E)\) with respect to parameters \(t\) and \(W_i\), it can be done in \(O(\log^3 n)\) time using \(O(n^3)\) processors for constructing a coarse vertex cover \(\mathcal{H}_i\) on \(V\) such that two vertices \(u\) and \(v\) are included in a home cover \(\mathcal{X}\) if \(d(u, v) \leq 2^{\log t - 1} W_i\) for fixed \(i\), where \(\mathcal{X} \in \mathcal{H}_i\).

**Proof.** By Algorithm 7, Step 1 can be finished in \(O(n \log t)\) time using \(O(n^3)\) processors. The details are as follows. First compute the matrix \(A^{[\log t] - 1}\) which requires \(O(n \log n)\) time using \(O(n^3)\) processors. Then construct the graph \(G_i(V, E^*)\) which requires \(O(1)\) time using \(O(n^3)\) processors. Step 2 can be done in \(O(\log^3 n)\) time using \(O(n^2/\log n)\) processors by the algorithm of Goldberg and Spencer [74]. The processing of Step 3 is more complicated. For every vertex \(u \in U(i, [\log t] - 1)\), first build an inverted tree \(T_i(u, [\log t])\) rooted at \(u\). Then check whether or not \(v \in T_i(u, [\log t])\) by checking \(a_i^{[u,v]} \leq 2^{\log t} W_i\) for every \(v \in V\). If it does, we find the parent \(p(u, v)\) of \(v\) in the tree, where \(p(u, v)\) is such a vertex \(w\) that \(a_i^{[u,w]} = a_i^{[u,v]} + wt((u, v))\). This can be done in \(O(\log n)\) time using \(O(n)\) processors. There are at most \(|U(i, [\log t] - 1)| \leq n\) trees. So, Step 3 requires \(O(\log n)\) time and \(O(n^2)\) processors. Step 4 can be done in \(O(1)\) time using \(O(n^2)\) processors. \(\square\)

**Theorem 8.6** For a weighted connected graph \(G(V, E)\) with non-negative weight, the spanner obtained by Algorithms 6 and 7 has size of \(O(\min\{m, n^2 T \log_{1+\epsilon}(\frac{W}{W_{min}})\})\) and factor of \(2t\), where \(\epsilon\) is a constant with \(0 < \epsilon < 1/2\).

**Proof.** The proof is similar to Theorem 8.5 and is omitted. \(\square\)

8.5 Conclusions

In this chapter, we first introduced the approximate \(t\)-spanner concept, and then presented simple parallel algorithms for finding such an approximate \(t\)-spanner on both unweighted and weighted graphs. The algorithms exhibit a trade-off between the running time and the factor of the spanner generated. Finally we presented an NC algorithm for finding a moderate \(2t\)-spanner on a weighted graph. The proposed algorithms are the first ones in the parallel context.
Parallel Algorithms for Edge-Coloring

9.1 Introduction

Let $G(V, E)$ be a simple graph with maximum vertex degree $\Delta(G)$ (or $\Delta$ for short), $|V| = n, |E| = m$. An edge coloring of $G$ is an assignment of colors to its edges such that all edges sharing a common vertex are assigned different colors. Furthermore, we wish to use as few colors as possible to color $G$. The minimum number of colors used is called chromatic index, denote by $\chi'(G)$. Vizing showed that $\Delta \leq \chi'(G) \leq \Delta + 1$ [171]. In fact, Vizing's proof implies $O(nm)$ sequential time algorithm with $\Delta + 1$ colors for the edge-coloring problem. However, Holey has shown that deciding whether a graph requires $\Delta$ or $\Delta + 1$ colors is NP-complete, even when restricted to the class of cubic graphs [82]. For a multigraph $G$, Shannon has shown that $\chi'(G) \leq 3\Delta/2$ [156].

A number of parallel algorithms for the edge-coloring problem have been proposed. Lev et al. [111] presented parallel edge-coloring algorithms with $\Delta$ colors for bipartite graphs. When $\Delta = 2^k$, their algorithm requires $O(\log \Delta \log n)$ time and $O(n\Delta)$ processors. Otherwise their algorithm requires $O(\log \Delta \log^2 n)$ time and $O(n\Delta)$ processors on the EREW PRAM. Gibbons et al. [71] and Gibbons and Ryutter [72] suggested algorithms for some other special graphs such as trees, outplanar graphs and Halin graphs. For trees, their algorithm requires $O(\log n)$ time and $O(n)$ processors; for outplanar graphs, their algorithm requires $O(\log^3 n)$ time and $O(n^2)$ processors; for Halin graphs, their algorithm requires $O(\log^2 n)$ time and $O(n)$ processors. All of their algorithms run in the CREW PRAM, and the number of colors used is $\Delta$. For planar graphs, Chrobak and Yung [28] presented an edge-coloring algorithm with $\max\{\Delta, 19\}$ colors. Their algorithm runs in $O(\log^2 n)$ time and uses $O(n)$ processors on the EREW PRAM. Later Chrobak and Nishizeki [27] improved the algorithm in [28] by reducing the number of colors to $\max\{\Delta, 8\}$. Their algorithm requires $O(\log^3 n)$ time and $O(n^2)$ processors. He [79] also discussed the edge-coloring problem with $\Delta$ colors for planar graphs, his algorithm has the same complexity as that of [28]. For general graphs, the pioneering work is due to Karloff and Shmoys [96]. They presented an edge-coloring parallel algorithm with $\Delta + 1$ colors which requires $O(\Delta^6 \log^3 n)$ time and $O(n^2 \Delta)$ processors on a COMMON CRCW PRAM, assume that the fastest known algorithm for
finding maximal independent sets [74] is used. They also presented a randomized $NC$
parallel algorithm with $\Delta + 20\Delta^{1/2+\epsilon}$ colors for edge-coloring, where $\epsilon \leq 1/4$, which
runs in $O(\log^{O(1)} n)$ expected time and uses $O(n^{O(1)})$ processors (independent of $\epsilon$).
For multigraphs, Upfal presented an $O(\log^2 n\Delta)$ time algorithm with $3\lceil \Delta/2 \rceil$ colors
by using $O(n\Delta)$ processors (appears in [96]). For the special class of multigraphs of
$\Delta = 3$, Karloff and Shmoys [96] presented an algorithm which runs in $O(\log n)$ time
and uses $O(n)$ processors only.

In this chapter we study the following three problems. The first one is to deal
with the edge-coloring update problem when a new vertex and the edges incident to
the vertex are added to an existing edge-coloring graph $G$ with $\Delta + 1$ colors. Let
the degree of this new vertex be at most $\Delta$. The proposed parallel algorithm for this
problem runs in $O(\Delta^{3/2} \log^3 \Delta + \Delta \log n)$ time using $O(n\Delta + \Delta^3)$ processors. Later we
use it as a basic subroutine for solving the second problem. The second one is to color
the edges of an arbitrary simple graph $G$ with $\Delta + 1$ colors, for which three parallel
algorithms are suggested: the first algorithm requires $O(\Delta^5 \log^4 n + \Delta^{5.5} \log^3 \Delta \log n)$
time and $O(n^2\Delta + n\Delta^3)$ processors, which is a slight improvement on Karloff and
Shmoys’ algorithm; the second algorithm has the same time and processor bounds as
the first one but the idea behind the algorithm is very useful to develop our third algorithm; the third algorithm requires $O(\Delta^4 \log^4 n + \Delta^{4.5} \log^3 \Delta \log n)$ time and
$O(n^2\Delta + n\Delta^3)$ processors, which improves by an $O(\Delta^{1.5})$ factor in time on Karloff
and Shmoys’ algorithm. The last one is to deal with the approximate edge-coloring
problem. For this problem, we present a fast parallel algorithm whose computational
complexity depends on the number of colors used. Unless otherwise specified, all our
algorithms and Karloff and Shmoys’ algorithm run in the COMMON CRCW PRAM.

9.2 Concepts and Notations

In this chapter we consider only a simple, connected, undirected graph $G(V, E)$, since
the proposed algorithm can be applied to each connected component (CC) of $G$ if $G$
is disconnected. Before we proceed, we introduce the well known Vizing Theorem.

Vizing’s Theorem [171] Any simple graph $G(V, E)$ can be edge-colored with $\Delta + 1$
colors.

Let $G$ be an edge-colored graph. A color $\alpha$ is said to be incident to a vertex $v$ if
there exists an edge of color $\alpha$ which is incident to $v$; otherwise, $\alpha$ is free at $v$. Since
there are $\Delta + 1$ colors available and the degree of any vertex is at most $\Delta$, there always
exists a free color at each vertex. The proof of Vizing’s Theorem relies on the two re-coloring techniques, chain and fan, which are explained below.

A graph $G$ is partially colored if some edges of $G$ are colored and some are not.
Consider the subgraph of $G$ induced by the edges of colors $\alpha$ and $\beta$. This subgraph
must consist of a set of vertex-disjoint paths and simple cycles. Therefore, exchanging
the colors $\alpha$ and $\beta$, for edges in such a path or such a cycle, yields a new partial
coloring. Consider such a path from $u$ to $v$, where $\alpha$ is free at $u$ and $\beta$ is free at $v$. If
there exists an uncolored edge \((w,u)\) in \(G\) such that \(\beta\) is free at \(w\), but \(\alpha\) is not, then after we exchange the colors on the path (called \(\alpha\beta\)-path of \(u\)), \(w\) and \(u\) will have a common free color \(\beta\), and \((w,u)\) can be colored with \(\beta\). This procedure is called chain re-coloring or inverting the \(\alpha\beta\)-path.

There are many definitions of fan. Here we use the definition by Misra and Gries [133]. Let \(u\) be a vertex called the fan center. A fan \(<v...w>\) of \(u\) is an ordered sequence of vertices that satisfies all the following conditions: (i) \(<v...w>\) is a non-empty sequence of distinct neighboring vertices of \(u\); (ii) the edge \((u,v)\) is uncolored, and the edge \((u,s)\) is colored for any vertex \(s\) other than \(v\) in the sequence; (iii) if \(s\) and \(t\) are two consecutive vertices of the sequence, denoted by \(s \leq t\), \((s\) is the immediate predecessor of \(t\) in the sequence), then the color of the edge \((u,t)\) is free at \(s\).

Given an uncolored edge \((u,v)\), the following procedure constructs a fan \(F\) of \(u\) that is maximal in that it cannot be extended.

**Procedure Construct\(\alpha\)-fan \(F\) of \(u\)**

\[
F := <v> ;
\]

/* \(F\) is a queue, and let \(w\) be the rear of \(F\) */

while there exists an \(s: (u,s) \in E, s \notin F\), and the color of \((u,s)\) is free at \(w\) do

add \(s\) to the rear of \(F\);

\(w := s\);

endwhile

Given a fan \(<v...w>\) centered at \(u\) as defined above, rotating this fan is to re-color edge \((u,s)\) with the color of edge \((u,t)\) for any pair of consecutive vertices \(s\) and \(t\) \((s \leq t)\) in the fan. The propose of rotating is to color the edge \((u,v)\). As a result, the edge \((u,w)\) becomes uncolored.

We now color an uncolored edge \((u,v)\) after finding a maximal fan of \(u\). Let \(<v...w>\) be a maximal fan of \(u\) and let \(\alpha\) be free at \(u\) and \(\beta\) be free at \(w\). The algorithm by Misra and Gries [133] is as follows:

**Step 1.** Exchange the colors on the \(\alpha\beta\)-path of \(u\) so that \(\beta\) becomes free at \(u\);

**Step 2.** There exists an \(s\) which satisfies \(s \in <v...w>, <v...s>\) is a fan of \(u\), and \(\beta\) is free at \(s\). Rotate fan \(<v...s>\) of \(u\) and assign color \(\beta\) to the edge \((u,s)\).

### 9.3 An Edge-Coloring Update Algorithm

Given an edge-colored graph \(G(V,E)\) with \(\Delta + 1\) colors, we consider the edge-coloring update problem when a new vertex \(v\) of degree \(K\) is added to \(G\) \((K \leq \Delta)\). It is not difficult to design an \(O(Km)\) time sequential algorithm for this problem. However, designing a fast parallel algorithm for it does seem difficult. In this section we present an algorithm for the problem which requires \(O(\Delta^{3/2} \log \Delta + \Delta \log n)\) time and \(O(n + \Delta + \Delta^3)\) processors. Our algorithm is composed of two stages. The first stage colors as many edges incident to \(v\) as possible without changing the original edge-coloring of \(G\). Thus, there will be no common free color at both \(v\) and \(u\) for any remaining uncolored edge \((u,v)\) after the first stage. In the second stage, a fan graph \(FG\) is constructed first.
Then, for an uncolored edge \((u,v)\), a maximal path on \(FG\) is found which corresponds to a maximal fan centered at \(v\), and the edge \((u,v)\) is colored by rotating this maximal fan. This process is repeated until all the other uncolored edges incident to \(v\) are colored.

Now we consider the first stage. Let \(v\) be the new vertex adjacent to vertices \(v_1,v_2,\ldots,v_K\) in \(G\). Firstly we construct a color request graph \(G_1(X,Y,E_1)\), where \(X = \{v_1,v_2,\ldots,v_K\}\), \(Y\) is the set of \(\Delta + 1\) colors, and edge \((v_i,\alpha)\) \(\in E_1\) if and only if \(\alpha \in Y\) is free at \(v_i\). Obviously, \(G_1\) is a bipartite graph and it can be constructed in \(O(1)\) time with \(O(K\Delta)\) processors. Secondly we find a maximal matching \(M\) in \(G_1\) using the algorithm by Israeli and Shiloach [91], which requires \(O(\log^3 \Delta)\) time and \(O(n^2)\) processors. For every edge \((v_i,\alpha)\) \(\in M\), we color the edge \((v,v_i)\) with \(\alpha\). After that, the two endpoints of any uncolored edge have no common free color.

The second stage is to color the remaining uncolored edges by applying the fan operation. For an uncolored edge \((v,v_i)\), we construct a maximal fan centered at \(v\). The construction of the fan seems to be “inherently” sequential, so, in their algorithm Karloff and Shmoys sequentially construct the maximal fan. Here we present another approach to constructing the maximal fan in parallel, which is the crucial component of the proposed algorithm. Let \(U = \{v_i \mid (v,v_i)\) is not colored\}, and \(N(v_i) = \{c \mid \text{color } c \text{ is free at } v_i\}\). Assume that \(U = \{u_1,u_2,\ldots,u_s\}\), where \(1 \leq s < K\). We construct an auxiliary directed graph \(G_2(V_2,E_2)\), (i.e., the \(FG\)), where \(V_2 = U \cup Y\), an edge \((\alpha,\beta)\) is included in \(E_2\) if there exists \(v_i\) such that edge \((v,v_i)\) is colored \(\alpha\) and \(\beta \in N(v_i)\), and each edge \((u_j,\gamma)\) belongs to \(E_2\), where \(u_j \in U\) and \(\gamma \in N(u_j)\), \(1 \leq j \leq s\). If an edge \((v,v_i)\) is colored \(\alpha\), then we call \(v_i\), denoted by \(f(\alpha)\), the image of \(\alpha\) and \(\alpha\) the source of \(v_i\), because they are uniquely determined by each other.

**Lemma 9.1** Let \(G_2\) be defined as above, then a maximal directed path starting at \(u_i\) corresponds to a maximal fan centered at \(v\), for an uncolored edge \((v,u_i)\), \(1 \leq i \leq s\).

**Proof.** Let \(P\) be a directed maximal path of \(G_2\) starting at \(u_i\) and ending at a color vertex \(\delta\) (i.e., \(\delta \in Y\)). Note that every color vertex \(\beta\) (except \(\delta\)) on \(P\) corresponds to a unique edge colored \(\beta\). Assume that \(\sigma\) and \(\gamma\) (\(\gamma \neq \delta\)) are two consecutive vertices on \(P\). Let edges \((v,x)\) and \((v,y)\) be colored \(\sigma\) and \(\gamma\) respectively. By the definition of \(G_2\), the color \(\gamma\) is free at vertex \(x\). Furthermore, the ending vertex \(\delta\) of \(P\) implies that either there is an edge \((v,v_i)\) colored \(\delta\) and each color in \(N(v_i)\) has already appeared in \(P\) or no edge \((v,v_j)\) incident to \(v\) is colored \(\delta\), \(j \neq i\). Therefore, if \(P = < u_i,\alpha_1,\alpha_2,\ldots,\alpha_t>\), then \(< u_i,x_1,x_2,\ldots,x_t>\) is a maximal fan, where \(\alpha_i\) is the color of edge \(\langle v,x_i\rangle\) and \(\alpha_i\) is free at \(x_{i-1}\) \(\langle x_0 = u_i\rangle\). On the other hand, let \(u_i,x_1,x_2,\ldots,x_t\) be a maximal fan centered at \(v\). Then color \(\alpha_j\) is free at \(x_{j-1}\) \(1 < j \leq t\), and \(\alpha_1\) is free at \(u_i\). Therefore, \(u_i,\alpha_1,\alpha_2,\ldots,\alpha_t\) is a directed path of \(G_2\) by the definition of \(G_2\). The path is also a maximal path. □

Now we color the uncolored edge \((v,u_i)\). We begin by finding a maximal directed path \(P\) starting at \(u_i\) in \(G_2\). This can be implemented by directly applying the algorithm of Goldberg et al [75]. Given such a path \(P\), let \(P'\) be the corresponding maximal fan \(< u_i \ldots w >\) centered at \(v\), \(\alpha\) be free at \(u_i\), and \(\beta\) be free at \(w\). We apply
§9.4 A New Edge-Coloring Algorithm

133

the rotating fan operation to color edge \((v, u_i)\). That is, invert the \(\alpha\beta\)-path of \(v\), update \(N(v_j)\) for every vertex \(v_j\) on the \(\alpha\beta\)-path, find the first vertex \(x\) in the fan such that \(\beta \in N(x)\), rotate the fan \(< u_i \ldots x >\), and re-color edge \((v, x)\) with color \(\beta\). In order to find \(x\), assume that the path \(P\) is represented by a linked list in which each vertex \(s\) has a pointer pointing to its immediate predecessor \(F(s)\). We then show how to find \(x\). Suppose the ending vertex of \(P\) is \(\beta\), we apply the prefix computation algorithm to broadcast \(\beta\) to all other vertices on \(P\). Let \(y (= f(\lambda))\) be the image of a color vertex \(\lambda\) on \(P\). Then we check whether \(\beta \in N(y)\) at each color vertex \(\lambda\) on \(P\). If it does, that means that there exists a vertex \(y\) on \(P'\) such that edge \((v, y)\) is colored \(\lambda\) and \(\beta\) is free at \(y\) in the corresponding fan. In this case, we set a special tag \(\text{change}(\lambda)\) to “1” for vertex \(\lambda\). Initially we set \(\text{change}(\lambda)\) to “0” for each color vertex \(\lambda\) on \(P\). There may be more than one \(\lambda\)s on \(P\) tagged “1”, we find such a vertex \(\varphi\) whose distance to \(u_i\) is the shortest among all tagged vertices by using prefix computation. Then broadcast \(\varphi\) along \(P\) toward to \(u_i\). When a vertex \(\lambda\) receives \(\varphi\), it sets \(\text{change}(\lambda) := “2”\). Note that the image of \(\varphi\) is the vertex \(x (= f(\varphi))\). Now \(< u_i \ldots x >\) becomes a fan centered at \(v\). A vertex \(s\) in \(< u_i \ldots x >\) if and only if its corresponding source \(\lambda\) on \(P\) has \(\text{change}(\lambda) = “2”\).

Having finished the above preprocessing, the fan \(< u_i \ldots x >\) can be characterized by a pair of free colors \((\alpha, \beta)\). We color the edge \((v, u_i)\) as follows. Construct a subgraph \(G_{\alpha,\beta}\) of \(G\) induced by the edges with colors \(\alpha\) and \(\beta\), find all CCs of \(G_{\alpha,\beta}\) invert the \(\alpha\beta\)-path to which \(u_i\) belongs, and rotate fan \(< u_i \ldots x >\). All the operations can be done in parallel. Thus, the edge \((v, u_i)\) is colored and the edge \((v, x)\) is colored with color \(\beta\).

**Theorem 9.1** Given an edge-colored graph \(G(V, E)\) with \(\Delta + 1\) colors and a new graph \(G'\) obtained from \(G\) by adding a new vertex of degree \(K (K \leq \Delta)\) to \(G\), the edge-coloring of \(G'\) requires \(O(K \sqrt{\Delta \log \Delta} \Delta + K \log n)\) time and \(O(n \Delta + \Delta^3)\) processors.

**Proof.** From the above discussion, we know that the proposed algorithm for this problem is composed of two stages. In the first stage, constructing the color request graph \(G_1\) needs \(O(1)\) time and \(O(K \Delta)\) processors, and finding a maximal matching in \(G_1\) requires \(O(\log^3 \Delta)\) time and \(O(K \Delta)\) processors [91]. In the second stage, constructing \(G_2\) requires \(O(1)\) time and \(O(\Delta^3)\) processors; finding a maximal directed path in \(G_2\) that corresponds to a maximal fan requires \(O(\sqrt{\Delta \log \Delta} \Delta)\) time and \(O(\Delta^2)\) processors [75]; the prefix computation on \(P\) requires \(O(\log \Delta)\) time and \(O(\Delta)\) processors; computing CCs of \(G_2\) and inverting the \(\alpha\beta\)-path require \(O(\log n)\) time and \(O(n \Delta)\) processors [159]. In addition, the second stage needs at most \(K\) iterations. \(\square\)

9.4 A New Edge-Coloring Algorithm

9.4.1 A simple version of the Karloff-Shmoys' algorithm

Now we begin to discuss the edge-coloring problem with \(\Delta + 1\) colors. The parallel algorithm by Karloff and Shmoys [96] is based on the original proof of Vizing's Theorem.
Here we give a simple parallel algorithm. The basic idea of our algorithm is the same as theirs. The difference is that ours is based on a new constructive proof of Vizing’s Theorem by Misra and Gries [133].

Before we proceed, we introduce some definitions first. If $G$ is partially colored, then a vertex $v$ is said to be active if there is at least an uncolored edge incident to $v$. Let $S$ be a vertex subset. Denote by $G_S$ the graph induced by $S$. Define an undirected graph $G^2(V, E^2)$ from $G(V, E)$, where $E^2 = E \cup \{(u, v) \mid$ there exists a vertex $w$ such that $(u, w) \in E$ and $(w, v) \in E\}$. This means that there is an edge in $G^2$ between vertices $u$ and $v$ if and only if there is a path of no more than two edges between $u$ and $v$ in $G$. Obviously $G^2$ can be constructed in $O(\log \Delta)$ time with $O(n^2 \Delta)$ processors.

Denote by $A$ the set of active vertices. The proposed algorithm consists of $O(\Delta^5 \log n)$ phases. In each phase we try to color as many uncolored edges as possible. However, there may exist edge-coloring conflicts, i.e., an edge may be assigned with different colors at the same time when the edge-coloring operations are executed in parallel. We later show that the edges incident to $\Omega(|A|/\Delta^4)$ active vertices can be colored in parallel without any edge-coloring conflict. We also know that each active vertex has at most $\Delta$ uncolored edges. Therefore, an $\Omega(1/\Delta^5)$ factor of the remaining uncolored edges will be colored in each phase. As a result, $O(\Delta^5 \log n)$ phases are enough to color all uncolored edges incident to active vertices. The detailed algorithm is as follows.

**Algorithm 1: COLOR\_EDGE\_A.**

**Step 0.** Initialization, i.e., $A := V$.

**Step 1.** Construct $G^2_A$ (induced by $A$), and find a maximal independent set $I$ of $G^2_A$ by using the algorithm of Goldberg and Spencer [74] which requires $O(\log^3 n)$ time and $O(n \Delta^2 / \log n)$ processors.

**Step 2.** For each vertex $v \in I$, choose an uncolored edge $e(v)$ incident to $v$. Denote by $M$ the set of the chosen uncolored edges.

**Step 3.** Set $em(v) := u$ for each edge $e(v) = (u, v) \in M$. If there exists a color $\alpha$ which is free at both $v$ and $em(v)$, then color the edge $(v, em(v))$ with $\alpha$, and remove $v$ from $I$. Let $I^* (\subseteq I)$ be the remaining vertex set. Construct a directed graph $G_2$ defined in Section 9.3 and find a maximal fan centered at $v$ for every vertex $v \in I^*$ in parallel.

**Step 4.** For each vertex $v \in I^*$, there is a maximal fan centered at $v$, characterized by a pair of colors $(\alpha(v), \beta(v))$, where $\alpha(v)$ is free at $v$ and $\beta(v)$ is free at the rear of this fan. Let $I_{\alpha, \beta} = \{v \mid v \in I^*$ and the maximal fan at $v$ is characterized by $(\alpha, \beta)\}$. Denote by $I^{**}$ the largest $I_{\alpha, \beta}$, i.e., $|I_{\alpha, \beta}| = \max_{\delta, \gamma \in \{\Delta + 1 \text{ colors}\}} |I_{\delta, \gamma}|$.

Let $u \prec v \prec w$ be a maximal fan at $u$, $\alpha$ be free at $u$, and $\beta$ be free at $w$. Suppose that $t$ is the first vertex in $u \prec v \prec w$ such that edge $(u, t)$ is colored with $\beta$. Then $\beta$ must be free at $t$’s immediate predecessor $s$ ($s \preceq t$). Assign $x(v) := t$, $y(v) := w$, and $z(v) := s$. Otherwise none of the edges incident to $v$ is colored with $\beta$. In this case, we set $x(v) := \emptyset$, $y(v) := w$, and $z(v) := \emptyset$. Before applying the algorithm by Misra and Gries to each vertex $v \in I^{**}$, let us recall the precondition of Step 2 in their algorithm.
The edge is colored with $\alpha$ or $\beta$.

The edge is colored with neither $\alpha$ nor $\beta$.

Color $\beta$ is free at this vertex.

**Figure 9.1:** Inverting the $\alpha\beta$-path of $v_2$ affects the free color of $w_1$ in a fan of $v_1$, and inverting the $\alpha\beta$-path of $v_3$ affects the free color of $z(v_2)$ in a fan of $v_2$.

It says that there exists an $s$ that satisfies $s < v \ldots w >, < v \ldots s >$ is a fan of $u$, and $\beta$ is free at $s$. However, after inverting the $\alpha\beta$-path of $v$ for all $v \in I^*$ in parallel, this precondition may not hold any more for all $v \in I^*$. For example, in Fig. 9.1, there are three maximal fans $< u_i \ldots w_i >$ at $v_i, i = 1, 2, 3$, characterized by a pair of colors $\alpha$ and $\beta$. Assume that $x(v_1) = \emptyset, y(v_1) = w_1, z(v_1) = \emptyset, x(v_2) \neq \emptyset, y(v_2) = w_2, z(v_2) \neq \emptyset, x(v_3) \neq \emptyset, y(v_3) = w_3, z(v_3) \neq \emptyset$. Now we invert the $\alpha\beta$-path starting at $v_i$ simultaneously, $i = 1, 2, 3$. As a result, the color $\beta$ is no longer free at both $w_1$ and $z(v_2)$, and the precondition of Step 2 of Misra and Gries' algorithm does not hold any more at $v_1$ and $v_2$. In order to avoid this situation, we construct a conflict graph $G_{con-flid}(I^*, E^*)$ such that $(u, v)$ is an edge of $E^*$ if and only if the precondition does not hold when we invert their $\alpha\beta$-paths of $u$ and $v$ in parallel. Let $I^{**}$ be a maximal independent set of $G_{con-flid}$. It is obvious that the precondition will hold after we invert the $\alpha\beta$-path of $v$ for all $v \in I^{**}$. $I^{**}$ can be calculated as follows. we first construct the subgraph $G_{\alpha, \beta}$ of $G$ induced by the edges colored $\alpha$ or $\beta$. Then we construct the graph $G_{con-flid}(I^{**}, E^{**})$ where $(u, v) \in E^{**}$ if and only if either $x(v) \neq \emptyset$ and $y(u) \neq \emptyset$. (In the rest of this section, we assume that $G_{\alpha, \beta}$ is full of edges colored for some color $\gamma$.)
belong to the same CC of $G_{\alpha, \beta}$, or $x(v)$ and $z(u)$ ($\neq \emptyset$) belong to the same CC of $G_{\alpha, \beta}$. In fact, there exists an edge between $u$ and $v$ in $G_{\text{conflict}}$ if and only if there is an $\alpha \beta$-path whose two endpoints are $x(v)$ and $y(u)$ or $x(v)$ and $z(u)$ respectively. Obviously $\Delta(G_{\text{conflict}}) \leq 3$, because there are at most three edges incident to vertex $v$ which are generated by $x(v)$, $y(v)$ and $z(v)$. A maximal independent set $I^{* * *}$ of $G_{\text{conflict}}$ can be found using the algorithm of Goldberg and Spencer [74].

Step 5. Apply Misra and Gries’ algorithm now, i.e., invert the $\alpha \beta$-path of $v$, rotate the fan centered at $v$, and color edge $(v, c)(v) \in M$ with color $\beta$ for each $v \in I^{* * *}$.

Step 6. If $v \in \mathcal{A}$ and all edges incident to $v$ have been colored, then $\mathcal{A} := \mathcal{A} - \{v\}$.

Step 7. If $\mathcal{A} \neq \emptyset$, goto Step 1.

Theorem 9.2 Given a simple undirected graph $G(V, E)$, $|V| = n$. The edge-coloring of $G$ by Algorithm 1 requires $O((\Delta^5 \log^4 n + \Delta^{5.5} \log n \log^3 \Delta)$ time and $O(n^2 \Delta + n \Delta^3)$ processors.

Proof. Let $|\mathcal{A}| = n_0$. Then, $|I| \geq \Omega(n_0 / \Delta^2)$, because $\Delta(G_{\mathcal{A}}) \leq \Delta^2$ and $I^* \subseteq I$. There are $\Delta^2$ pairs of colors, so, $|I^{* * *}| \geq |I|/\Delta^2 = \Omega(n_0 / \Delta^4)$. We also know that $\Delta(G_{\text{conflict}}) \leq 3$. Thus, $|I^{* * *}| \geq |I^{* * *}|/|A| \geq \Omega(n_0 / \Delta^4)$. That means, $\Omega(n_0 / \Delta^4)$ uncolored edges can be colored in one iteration of Steps 1 to 7 of Algorithm 1. Initially $n_0 = n$. All other operations in Algorithm 1 can be done in $O(\log^3 (\Delta n))$ time with $O(n \Delta)$ processors except that (i) the construction of $G_2$ requires $O(1)$ time and $O(\Delta^3)$ processors and finding a maximal fan in $G_2$ requires $O(\sqrt{\Delta \log^3 \Delta})$ time and $O(\Delta^2)$ processors for each $v \in I^*$; (ii) the constructions of $G_{\mathcal{A}}^2$ and $G_{\text{conflict}}$ require $O(\log \Delta)$ time and $O(n^2 \Delta)$ processors, and $O(1)$ time and $O(n^2)$ processors, respectively. The number of iterations of Steps 1 to 7 is $O(\Delta^5 \log n)$, which has been shown at the beginning of this section. So, the proposed algorithm requires $O(\Delta^5 \log^4 n + \Delta^{5.5} \log^3 \Delta \log n)$ time and $O(n^2 \Delta + n \Delta^3)$ processors. □

9.4.2 An improved edge-coloring algorithm

Another edge-coloring algorithm that has the same time and processor complexities as Algorithm 1 is introduced first, and then an improved edge-coloring algorithm that uses part of the algorithm as its subroutine is presented. The basic idea of the proposed algorithm is as follows. We first use more than $\Delta + 1$ colors to color the edges in $G$ quickly. Actually, we use $3 \lceil \Delta / 2 \rceil$ colors to color the edges of $G$. We then remove all extra colors from the colored graph until $\Delta + 1$ colors are left. Consequently, we obtain a partially edge-colored graph with $\Delta + 1$ colors. Finally we color the remaining uncolored edges phase by phase. In each phase we only color those uncolored edges which were colored with the same color. Denote by $M$ the uncolored edge set. It is obvious that $M$ is a matching of $G$. We try to choose as many edges in $M$ as possible such that one of the endpoints of each chosen edge serves as the center of a maximal fan and no edge-coloring conflict will occur when the rotating fan operation is applied to each of the fans in parallel. In order to choose the edges, we construct an auxiliary undirected graph $G_3(V_3, E_3)$ where a vertex of $V_3$ represents an edge in $M$, i.e., $V_3 = M$. 
Let $e_1 = (u_1, v_1) \in M$ and $e_2 = (u_2, v_2) \in M$. An edge $(e_1, e_2)$ is added to $E_3$ if and only if the distance between an endpoint of $e_1$ and an endpoint of $e_2$ in $G$ is no more than 2. The detailed algorithm is as follows.

**Algorithm 2: COLOR\_EDGE\_2**

**Step 1.** Check whether $G$ is an Eulerian graph. If not, add a new vertex $v$ to $G$, and add an edge between $v$ and any vertex with odd degree in $G$. As a result, we obtain an Eulerian graph $G'$.

**Step 2.** Apply Upfal's edge-coloring algorithm for multigraphs [96] to color $G'$ with $3[\Delta/2]$ colors.

**Step 3.** Delete all colored edges incident to $v$ and calculate the number of edges in $G$ colored with $\alpha$, $\text{count}(\alpha)$, for each color $\alpha$. Let $q$ be the number of colors used. Sort the colors in increasing order by their counting numbers, and assign each color $\alpha$ a unique index $\text{index}(\alpha)$ between 1 and $q$ such that $\text{index}(\alpha) > \text{index}(\beta)$ if $\text{count}(\beta) > \text{count}(\alpha)$.

**Step 4.** While $q > \Delta + 1$ do /* repeat the following Step 5 to Step 11. */

**Step 5.** Remove the color from all the edges in $M$ where $M = \{ e \mid e$ is colored with color indexed $q \}$.

**Step 6.** Repeat the following Step 6 to Step 10. */

**Step 6.** If both endpoints of an uncolored edge in $M$ share a free valid color (colors indexed between 1 and $\Delta + 1$ are considered to be valid), then assign the edge with the free color. Let $M^*$ be the remaining uncolored edge set, construct $G_3$ for $M^*$.

**Step 7.** Find a maximal independent set $IE$ of $G_3$. Note that each vertex in $IE$ is an edge of $G$. For each vertex (an edge in $G$) in $IE$, choose one endpoint of the edge as the fan center. Denote by $I^*$ the set of fan centers, find a maximal fan $< u \ldots w >$ at $v$ by constructing a directed graph $G_2$ for each $v \in I^*$ in parallel.

**Step 8.** The same as Step 4 of Algorithm 1.

**Step 9.** The same as Step 5 of Algorithm 1.

**Step 10.** $M := M^* - \{ \text{all colored edges in Step 9} \}$

until $M = \emptyset$;

**Step 11.** $q := q - 1$;

endwhile.

**Lemma 9.2** Given a fixed $\theta_0$, the number of iterations of Steps 6—10 in Algorithm 2 is $O(\Delta^4 \log n)$.

**Proof.** Initially $|M| \leq |n/2|$ because $M$ is a matching of $G$. By the construction of $G_3$, $\Delta(G_3) \leq 2\Delta^2$. So, the size of any maximal independent set of $G_3$ is at least $|M|/(2\Delta^2 + 1)$, i.e., the number of maximal fans is at least $|M|/(2\Delta^2 + 1)$. There are $\Delta(\Delta - 1)/2 < \Delta^2/2$ pairs of colors for the maximal fans. Therefore, in each iteration of Steps 6-10, at least $|I^*| \geq |M|/(2\Delta^2 + 1) \geq |M|/(3\Delta^4)$ fans are rotated, which means $\Omega(|M|/\Delta^4)$ uncolored edges are colored. Since the size of $M$ reduces by a factor
of \((1 - 1/(3\Delta^4))\) in each iteration, coloring all edges in \(M\) requires \(O(\Delta^4 \log n)\) iterations of Steps 6-10. □

**Theorem 9.3** Given a simple undirected graph \(G(V,E), |V| = n\), the edge-coloring of \(G\) with \(\Delta + 1\) colors by Algorithm 2 requires \(O(\Delta^5 \log^4 n + \Delta^5.5 \log^3 \Delta \log n)\) time and \(O(n^2 \Delta + n \Delta^3)\) processors.

**Proof.** By Algorithm 2, we know that all operations in Steps 1-3 require at most \(O(\log^3 (n\Delta))\) time and \(O(n\Delta)\) processors. We also know that we need to execute the **while** loop \(t\) times, where \(t = (\Delta + 3)/2\) if \(\Delta\) is odd, otherwise \(t = \Delta/2 - 1\). Therefore we need to execute the **repeat** loop a total of \(O(\Delta^5 \log n)\) times by Lemma 9.2. The construction of \(G_3\) in Step 6 requires \(O(\log \Delta)\) time and \(O(n\Delta)\) processors. Step 7 can be done in \(O(\log^3 n)\) time using \(O(n\Delta^3)\) processors because finding a maximal independent set in \(G_3\) needs \(O(\log^3 n)\) time and \(O(n\Delta^2 / \log n)\) processors; constructing \(G_2\) and finding a maximal fan in \(G_2\) require \(O(\sqrt{\Delta \log^3 \Delta})\) time and \(O(\Delta^2)\) processors \([75]\) and \(O(1)\) time and \(O(\Delta^3)\) processors, respectively. According to Algorithm 1, Steps 8 and 9 can be finished in \(O(\log^3 n)\) time using \(O(n\Delta)\) processors. Thus, the entire algorithm requires \(O(\Delta^5 \log^4 n + \Delta^{5.5} \log^3 \Delta \log n)\) time and \(O(n^2 \Delta + n \Delta^3)\) processors. □

Then we have the following lemma.

**Lemma 9.3** Given two simple undirected graphs \(GA_1(VA_1, EA_1)\) and \(GB_2(VB_2, EB_2)\), \(VA_1 \subseteq V\) and \(VB_2 \subseteq V\), \(|V| = n\), \(EA_1 \cap EB_2 = \emptyset\). Assume that \(GA_1\) and \(GB_2\) are the edge-colored graphs with \(\Delta(GA_1) + 1\) and \(\Delta(GB_2) + 1\) colors, respectively. Let \(C_1\) and \(C_2\) be their color sets, where \(C_1 \cap C_2 = \emptyset\). Then edge-coloring the graph \(GA_1 \cup GB_2 = (VA_1 \cup VB_2, EA_1 \cup EB_2)\) with \(\Delta_{AB} + 1\) colors requires \(O(\Delta_{AB}^{1.5} \log^3 \Delta_{AB} \log n + \Delta_{AB}^4 \log^4 n)\) time and \(O(n^2 \Delta_{AB} + n \Delta_{AB}^3)\) processors, where \(\Delta_{AB} = \Delta(GA_1) + \Delta(GB_2)\).

**Proof.** Because \(C_1 \cap C_2 = \emptyset\), the edges in graph \(GA_1 \cup GB_2\) have been colored with \(|C_1 \cup C_2| = \Delta(GA_1) + \Delta(GB_2) + 2\) colors. Now we remove a color \(\alpha\) from the color set \(C_1 \cup C_2\), and recolor those edges that were colored with \(\alpha\) in \(GA_1 \cup GB_2\). The remaining task is to color the uncolored edges by the colors in \(C_1 \cup C_2 - \{\alpha\}\), and such an edge-coloring is valid by Vizing’s Theorem. The implementation of such an edge-coloring can be done by Steps 6-10 of Algorithm 2. Now let us analyze its computational complexity. The number of iterations of Steps 6-10 of Algorithm 2 is \(O(\Delta_{AB}^4 \log n)\) by Lemma 9.2. Finding a maximal independent set requires \(O(\log^3 n)\) time and \(O(n \Delta_{AB}^2)\) processors. The constructions of \(G_2\) and \(G_{conflict}\) require \(O(1)\) time and \(O(n \Delta_{AB}^3)\) processors, and \(O(\log \Delta_{AB})\) time and \(O(n^2 \Delta_{AB})\) processors, respectively. Finding a maximal fan for \(G_2\) requires \(O(\Delta_{AB}^{1/2} \log^3 \Delta_{AB})\) time and \(O(\Delta_{AB}^2)\) processors \([75]\). Consequently, coloring all uncolored edges in \(GA_1 \cup GB_2\) requires \(O(\Delta_{AB}^{1.5} \log^3 \Delta_{AB} \log n + \Delta_{AB}^4 \log^4 n)\) time and \(O(n^2 \Delta_{AB} + n \Delta_{AB}^3)\) processors. □
Having Algorithm 2 and Lemma 9.3, we now present an improved parallel algorithm for the edge-coloring problem. The main idea is to decompose $G$ into many edge-disjoint subgraphs and to color and merge the subgraphs in parallel.

**Algorithm 3:** \texttt{COLOR\_EDGE\_3}

**Step 1.** Form a bipartite graph $G_B(X', Y', E_B)$ as follows. If $G$ is not an Eulerian graph, then add a new vertex $v$ and make it adjacent to all vertices of odd degrees. Find an Eulerian tour of the augmented graph, and view the tour as a directed tour by the algorithm due to Atallah and Vishkin \cite{8}. Let $X' = \{v \mid v \in V\}$ and $Y' = \{v'' \mid v \in V\}$. For each edge $(u, v)$ traversed from $u$ to $v$, add an undirected edge $(u', v'')$ to $E_B$. As a result, $G_B$ is a bipartite graph with maximum degree $\lceil \Delta(G)/2 \rceil$.

**Step 2.** Color the edges of $G_B$ with $\lceil \Delta(G)/2 \rceil$ colors, using the optimal algorithm of Lev \textit{et al.} \cite{111}.

**Step 3.** Let $M_i (\subseteq E_B)$ be a set of edges colored $c_i$, where $c_i$ is a color. Construct a subgraph $SG_i(V(M_i), E_i)$ as follows: for each edge $(u', v'') \in M_i$, add an edge $(u, v)$ into $E_i$. Since $M_i$ is a matching in $G_B$, $\Delta(SG_i) \leq 2$. Thus, $SG_i$ is a collection of disjoint paths and simple cycles. Now $G$ is decomposed into subgraphs $SG_0, SG_1, \ldots, SG_{\lceil \Delta/2 \rceil - 1}$.

**Step 4.** $p := \lceil \Delta/2 \rceil - 1$;

**Step 5.** for each $i : 0 \leq i \leq \lceil \Delta/2 \rceil - 1$ pardo

- color graph $SG_i$ using the colors in a color set $C_i$, $|C_i| = 3$,
- and $C_i \cap C_j = \emptyset$ if $i \neq j$; $G_i^{(0)} := SG_i$; $C_i^{(0)} := C_i$;

endfor;

**Step 6.** for $k := 0$ to $\lceil \log \Delta - 1 \rceil$ do

for each $i : 0 \leq i \leq \lfloor p/2 \rfloor$ pardo

- arbitrarily select a color $\alpha_i$, decolor those edges colored with $\alpha_i$ in graph $G_i^{(k+1)} := G_i^{(k)} \cup C_i^{(k)}$ where $\alpha_i \in C_i^{(k)}$;
- Let $M$ be the uncolored edge set of $G_i^{(k+1)}$, color the edges in $M$ with the colors from $C_i^{(k+1)} := C_i^{(k)} \cup C_i^{(k)} - \{\alpha_i\}$ by Lemma 9.3.

endfor;

if $p$ is even then

- $oldcount := p$; $p := p/2 + 1$;
- $G_p^{(k+1)} := G_{oldcount}^{(k)}$;
- /* the unpaired subgraph is assigned by a new index */

else $p := \lfloor p/2 \rfloor$;

endfor.

**Theorem 9.4** Given a simple undirected graph $G(V, E)$ with $|V| = n$, edge-coloring $G$ with $\Delta + 1$ colors by Algorithm 3 requires $O(\Delta^{1.5} \log^3 \Delta \log n + \Delta^4 \log^4 n)$ time and $O(n^2 \Delta + n \Delta^3)$ processors.

**Proof.** By Algorithm 3, we know that Steps 1-5 require at most $O(\log^3(n \Delta))$ time and $O(n \Delta)$ processors \cite{8, 111}. Now we analyze the computational complexity of
Step 6. Given a fixed \( k = k_0 \), we know that \( \Delta(G^{(k)}_i) \leq 2^{(k_0+1)} \), \( 0 \leq i \leq p \). So, edge-coloring graph \( G^{(k_0)}_i \) requires \( O((2^{(k_0+1)})^{1.5 \log^3 (2^{(k_0+1)})} \log n + (2^{(k_0+1)})^{4 \log^4 n}) \) time and \( O(n^2 2^{(k_0+1)} + n(2^{(k_0+1)})^{3}) \) processors by Lemma 9.3. Therefore, Step 6 requires \( \sum_{k=0}^{[\log \Delta - 1]} O((2^{(k+1)})^{1.5 \log^3 (2^{(k+1)})} \log n + (2^{(k+1)})^{4 \log^4 n}) = O(\Delta^{1.5} \log^3 \Delta \log n + \Delta^4 \log^4 n) \) time and \( O(n^2 \Delta + n \Delta^3) \) processors. \( \square \)

9.5 An Approximate Edge-Coloring Algorithm

In this section we study the approximate edge-coloring problem, that is, color the edges of \( G \) with \( \max \{2^{i-1} \times ([\Delta/2^{i-1}] + 2), \ 2^{i-1} \times ([\Delta/2^{i-1}] + 3) \} \geq \Delta + 1 \) colors for any fixed integer \( i \), \( 0 \leq i \leq [\log \Delta] - 1 \), such that all edges sharing a common vertex have different colors. We suggest a parallel algorithm for this problem, based on a graph decomposition technique. Our algorithm requires \( O((\Delta/2^{i-1})^{1.5 \log^3 \Delta \log n} + (\Delta/2^{i-1})^{4 \log^4 n}) \) time and \( O(n^2 \Delta/2^{i-1} + n(\Delta/2^{i-1})^{3}) \) processors with fixed \( i \). The most important is that we have found that there exists a kind of trade-off between the number of colors used and the amount of work to assign such an edge-coloring. That means, the fewer the number of colors used (\( \geq \Delta + 1 \)) to edge-color \( G \), the more time needed for such edge-coloring. Consequently, the approximate edge-coloring with 2.5 \( \Delta \) colors can be done in \( O(\log n \log \Delta) \) time using \( O(m + n) \) processors on a CREW PRAM. It is already known that edge-coloring \( G \) with \( 3[\Delta/2] \) colors requires \( O(\log^3 n) \) time and \( O(n^{3}) \) processors on a CREW PRAM [96]. In particular, edge-coloring \( G \) with \( \Delta + \sqrt{\Delta} \) colors only requires \( O(\Delta^{2.25} \log^3 \Delta \log n + \Delta^2 \log^4 n) \) time and \( O(n^2 \sqrt{\Delta} + n^2 \Delta^2) \) processors on a COMMON CRCW PRAM if Algorithm 3 is used as a subroutine.

In the following the graph decomposition concept is introduced, and an algorithm for the decomposition is also presented. An approximate edge-coloring algorithm based on the decomposition is devised later.

9.5.1 Graph decomposition

Given an Eulerian graph \( G \), the Eulerian decomposition is to partition \( G \) into edge-disjoint simple cycles. If \( G \) is not an Eulerian graph, add a new vertex \( v \) to it, and add an edge between \( v \) and each of odd degree vertices in \( G \) so that the resulting graph \( G' \) is an Eulerian graph. Let \( G \) (or \( G' \)) be an Eulerian graph. We first partition \( G \) into two edge-disjoint subgraphs \( G_1 \) and \( G_2 \) such that \( \Delta(G_1) = \Delta(G_2) \). It can be easily done by traversing an Euler tour and alternatively assigning every edge with red and blue colors during the Eulerian traversal. Let \( G_1 \) be the subgraph induced by all red edges, and \( G_2 \) be the subgraph induced by all blue edges. As a result, \( \Delta(G_1) = \Delta(G_2) \) and \( |E(G_1)| = |E(G_2)| \). Obviously, for an arbitrary simple graph \( G \), we can partition \( G \) into two edge-disjoint subgraphs \( G_1(V, E_1) \) and \( G_2(V, E_2) \) such that

\[
\Delta(G_1) \leq [\Delta(G)/2] \quad \text{and} \quad \Delta(G_2) \leq [\Delta(G)/2] + 1
\]  

(9.1)

However, it is a little more expensive to obtain this partition by Eulerian traversal in the parallel computing environment. Here we introduce an efficient way to partition
§9.5 An Approximate Edge-Coloring Algorithm

$G$ into $G_1$ and $G_2$. As Fürer and Raghavachari noted [60], we don’t need to find an Eulerian traversal in $G'$ to achieve the above partition. Instead, we find all edge-disjoint simple cycles in $G'$ initially, then we assign the edges on every simple cycle by the colors blue and red alternatively. The details are as follows. For an even simple cycle, we just arbitrarily select a vertex on it as the starting point and assign the edges with red and blue alternatively starting from the point. The odd cycles can be dealt with by applying the following simple rule: if two odd cycles $C_1$ and $C_2$ share a starting point $v$, $C_1$ (or $C_2$) is colored by red (blue) and blue (red) alternatively, starting from $v$. Now we have

**Lemma 9.4** Given a graph $G(V, E)$, partitioning $G$ into two edge-disjoint subgraphs $G_i(V, E_i)$ such that $\Delta(G_1) = \lceil \Delta(G)/2 \rceil$, $\Delta(G_2) = \lceil \Delta(G)/2 \rceil + 1$, $E_i \subseteq E$, $E_1 \cap E_2 = \emptyset$, $i=1,2$, can be done in $O(m + n)$ sequential time, or in $O(log n)$ time using $O(m + n)$ processors on a CREW PRAM.

**Proof.** It is easy to prove that the decomposition can be implemented in $O(m + n)$ sequential time, because the time complexity of sequential Eulerian traversal is within this bound. Now we prove this lemma by presenting an algorithm to implement the partition.

First of all, we calculate the vertex degree $d(v)$ for every vertex $v$ in $G$. Assume that $G$ is an Eulerian graph, if not, we use the above method to make it become an augmented Eulerian graph $G'$. Then, we apply the first part of Atallah and Vishkin’s algorithm for finding Euler tour [8] to find all edge-disjoint simple cycles in $G$ or $G'$, which can be implemented in $O(log n)$ time using $O(m + n)$ processors on a CREW PRAM. Note that we only make use of the first part of their algorithm, while the dominant computational complexity in their algorithm is of finding a spanning tree which we don’t need.

Having done the above preprocessing, the partition algorithm is then shown below, which consists of the following three steps.

Step 1 is to delete the new vertex and new added edges from the simple cycles if the original graph $G$ is not an Eulerian graph. As a result, $G$ is partitioned into simple cycles and simple paths.

Step 2 is to color all paths. For each simple path, we select one of its two endpoints as the starting point for coloring. Let $P$ be such a path with the two endpoints $v$ and $u$. Initially we set $T(v) := 0$ for each vertex $v$ in $G$. If $d(v) = \Delta$ and $d(u) < \Delta$, we select $v$ as the starting point, and color the path with blue first. If $d(v) = \Delta$ and $d(u) = \Delta$, we select $v$ as the starting point and color the path with blue first. Then we calculate the length of $P$, if the length of $P$ is even, we set $T(u) := 1$. If both $d(v) < \Delta$ and $d(u) < \Delta$, we arbitrarily select one of them as the starting point and use the blue color first.

Step 3 is to color all simple cycles. For each simple cycle $C$, regardless of odd or even length, arbitrarily select a vertex $v$ on it as the starting point to color. As a result, each cycle $C$ is associated with a tuple $(v, C)$. Then sort the tuples by key $v$. After sorting, color the edges on all even cycles with colors blue and red alternatively, and delete the tuples associated with the even cycles from the sorted sequence. It can be
implemented by compressing the sequence using the parallel prefix computation. Then the remaining tuples in the sequence correspond to odd cycles. The rest is to assign the colors red and blue to the edges on odd cycles. Let \( d(v) = \Delta \) and \( v \) be the starting point of \( k \) odd cycles, we pair the \( k \) odd cycles into \( \lfloor k/2 \rfloor \) pairs, and at most one of them is unpaired. For each two paired cycles, we color one of them by red first and another by blue first. If \( k \) is odd, which color to be used first for the unpaired cycle is determined by \( T(v) \). If \( T(v) = 1, v \) is one of the two endpoints of a path, and the edge incident to \( v \) on the path is already colored by red. In order to satisfy Condition 9.1, the unpaired cycle is colored with blue first. Otherwise the unpaired cycle is colored with red first. The details are as follows.

After finishing compressed, we calculate the number of times of \( v \) is specified as the starting point in the remaining tuples. Denote by \( num(v) \) the number. Obviously, all \( num(v) \) can be figured out in \( O(\log n) \) time using \( O(n/\log n) \) processors by prefix computation. Let \( l \) be the length of the remaining tuples. For two adjacent tuples

\[ (v'_{i}, C'_{i}) \text{ and } (v'_{i+1}, C'_{i+1}) \]

if \( v'_{i} \neq v'_{i+1}, num(v'_{i}) \) is odd, and \( T(v'_{i}) = 1 \), we will color \( C'_{i} \) with blue first and delete its associated tuple from the sequence, \( 1 \leq i \leq l - 1 \).

Then we compress the sequence again. Now we color the finally remaining tuples as follows: for each of them, if the rank (or index) of \( (v, C) \) in the sequence is odd, we color \( C \) starting from \( v \) by red first, otherwise we color \( C \) by blue first. Coloring all edges on a cycle with length at most \( n \) by two colors can be done in \( O(\log n) \) time using \( O(n) \) processors. Sorting \( m \) elements can be done in \( O(\log m) \) time using \( O(m) \) processors by the algorithm due to Cole \([30]\). As a result, \( G_{1} \) is a subgraph induced by all edges colored red, and \( G_{2} \) is a subgraph induced by all edges colored blue. It is not difficult to show that the above partition satisfies that \( \Delta(G_{1}) \leq \lceil \Delta(G)/2 \rceil \) and \( \Delta(G_{2}) \leq \lfloor \Delta(G)/2 \rfloor + 1 \).

9.5.2 An algorithm for approximate edge-coloring

Let \( G_{2j} \) and \( G_{2j+1} \) be two subgraphs induced from \( G_{j} \) after a partition satisfying Condition 9.1, \( 1 \leq i \leq \lceil \log \Delta \rceil - 1 \), and \( 2^{i-1} \leq j \leq 2^{i}-1 \). \( G_{1} = G \) initially. Now we describe an algorithm of approximate edge-coloring \( G \) with

\[
\max \{2^{i-1} \times (\lceil \Delta/2^{i-1} \rceil + 2), 2^{i-1} \times (\lceil \Delta/2^{i-1} \rceil + 3)\} \text{ colors, } 0 \leq s \leq \lceil \log \Delta \rceil - 1.
\]

The algorithm consists of two phases. In the first phase we partition \( G \) into a number of edge-disjoint subgraphs and each partition satisfies the condition of Lemma 9.4. In the second phase, we apply Algorithm 3 for each subgraph in the final partition. The detailed algorithm is presented as follows:

Algorithm 4: APPRO_COLOR\_EDGE

Initially \( G_{1} := G \);

Step 1. for \( i := 1 \) to \( s-1 \) do

\[ j := 2^{i-1} \text{ to } 2^{i} - 1 \text{ pardo} \]

partition \( G_{j} \) into two subgraphs \( G_{2j} \) and \( G_{2j+1} \)

endfor
endfor;
Step 2. \textbf{for} \quad i := 2^{s-1} \text{ to } 2^s - 1 \textbf{ pardo}
\quad color \ G_i \ \text{with } \max\{\lceil \Delta/2^{s-1} \rceil + 2, \lfloor \Delta/2^{s-1} \rfloor + 3\} \text{ colors}
\quad \text{by Algorithm 3 in Section 9.4.}
\textbf{endfor}.

Note that the edges of each subgraph are colored with the colors coming from different color sets. In the following we analyze the number of colors used in Algorithm 4. Let

\[ 2^k \leq \Delta < 2^{k+1}, \]  
(9.2)

then,

\[ \Delta = a_k 2^k + a_{k-1} 2^{k-1} + \ldots + a_2 2^2 + a_1 2 + a_0 \]  
(9.3)

where \( a_i = 0 \) or \( a_i = 1 \), \( 0 \leq i \leq k \). Let

\[ \phi(i, \Delta) = a_i 2^i \]  
(9.4)

where \( 0 \leq i \leq \lfloor \log \Delta \rfloor \). It is obvious that either \( \phi(i, \Delta) = 2^i \) or \( \phi(i, \Delta) = 0 \), because \( a_i \) equals either 0 or 1 by Equation 9.3. Define

\[ \lambda(i, k) = a_i 2^i + a_{i+1} 2^{i+1} + \ldots + a_{k-1} 2^{k-1} + a_k 2^k \]  
(9.5)

where \( 0 \leq i \leq \lfloor \log \Delta \rfloor \), and

\[ \delta(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise.} \end{cases} \]  
(9.6)

Now we re-write \( \Delta \) as

\[ \Delta = \lambda(0, i) + \lambda(i + 1, k), \quad 0 \leq i \leq k - 1. \]  
(9.7)

\textbf{Lemma 9.5} \ 2^k \leq \lambda(i, k) \leq 2^{k-i+1}, \ \text{if } k > 0 \ \text{and } i \leq k.

\textbf{Lemma 9.6} \ \lambda(i, k) \ \text{mod } 2^i = 0, \ \text{for all } 0 \leq s \leq i \leq k \ \text{and } k > 0.

\textbf{Lemma 9.7} \ After having \( s \) partitions for \( G(V, E) \) with maximum vertex degree \( \Delta \), the maximum vertex degree \( \Delta(G_j) \) of subgraph \( G_j \) in the final partition satisfies the following inequality.

\[ \Delta(G_j) \leq \max\{\lceil \Delta/2^s \rceil + 1, \lfloor \Delta/2^s \rfloor + 2\} \]  
(9.8)

where \( 2^{s-1} \leq j \leq 2^s - 1, \ 1 \leq s \leq \lfloor \log \Delta \rfloor - 1. \)

\textbf{Proof.} \ We prove it by induction. When \( s = 1 \), by Lemma 9.4, Inequality 9.8 holds. Assume that it holds when \( s = i \). Now we prove that it also holds when \( s = i + 1 \). Let
$G''$ be the subgraph after the $i$th partition. By the inductive assumption, we have

$$\Delta(G'') \leq \max\left\{\left\lceil \frac{\Delta}{2^{i-1}} \right\rceil, \left\lceil \frac{\Delta}{2^{i}} \right\rceil + 2\right\}$$

Now we distinguish the following two cases:

(i) If $\Delta(G'') = \left\lceil \frac{\Delta}{2^{i}} \right\rceil + 1$, i.e.,

$$\Delta(G'') = \left\lceil \frac{\lambda(i,k)}{2^{i} \Delta} + \frac{\delta}{2} \right\rceil + 1 = \frac{\lambda(i,k)}{2^{i+1}} + \frac{\delta(\lambda(i,k) - 1)}{2} + 1,$$

by Lemma 9.4, we partition $G''$ into two subgraphs $G'_1$ and $G''_2$, which are the subgraphs formed by the edges colored red and blue respectively. Therefore

$$\Delta(G'_1) \leq \left\lceil \frac{\Delta(G'_1)}{2} \right\rceil = \frac{\lambda(i,k)}{2^{i+1}} + \frac{\delta(\lambda(i,k) - 1)}{2} + 1.$$

and

$$\Delta(G''_2) \leq \left\lceil \frac{\Delta(G''_2)}{2} \right\rceil + 1 = \frac{\lambda(i,k) + \delta \lambda(\lambda(i,k) - 1)}{2} + \frac{\delta(\lambda(i,k) - 1)}{2} + 1.$$

(ii) If $\Delta(G'') = \left\lceil \frac{\Delta}{2^{i}} \right\rceil + 1$, the proof is similar to (i) and is omitted. □

**Theorem 9.5** Given a graph $G(V,E)$ with maximum vertex degree $\Delta$, edge-coloring $G$ with $\lambda(s-1,k) + \max\{3 \ast 2^{i-1}, (2 + \delta(\lambda(0, s-2)/2^{i-1})) \ast 2^{i-1}\}$ colors can be done in $O((\Delta/2^{i-1}) \ast 4.5 \log^3 \Delta \log n + (\Delta/2^{i-1}) \ast 4 \log^4 n)$ time using $O(n^2 \Delta/2^{i-1} + n(\Delta/2^{i-1})^3)$ processors on the COMMON CRCW PRAM, where $0 \leq s \leq \left\lceil \log \Delta \right\rceil - 1$.

**Proof.** In order to make the maximum vertex degree $\Delta(G_j) \leq \max\{\left\lceil \Delta/2^{i-1} \right\rceil + 1, \left\lceil \Delta/2^{i-1} \right\rceil + 2\}$ for every subgraph $G_j$, $2^{i-1} \leq j \leq 2^{i} - 1$, we need $s-1$ partition iterations, and each partition requires $O(\log n)$ time and $O(m + n)$ processors by Lemma 9.4. So all partitions require $O(s \log n)$ time and $O(m + n)$ processors. After finishing Step 1 of Algorithm 4, there are $2^{i-1}$ subgraphs with maximum degree $\Delta(G_j)$, so the number of colors used is $\max\{2^{i-1} \ast (\left\lceil \Delta/2^{i-1} \right\rceil + 2), 2^{i-1} \ast (\left\lceil \Delta/2^{i-1} \right\rceil + 3)\}$

$$\leq \lambda(s-1,k) + \max\{3 \ast 2^{i-1}, (2 + \delta(\lambda(0, s-2)/2^{i-1})) \ast 2^{i-1}\},$$

where $0 \leq s \leq \left\lceil \log \Delta \right\rceil - 1$. □

**Corollary 9.1** Given a graph $G(V,E)$ with maximum vertex degree $\Delta$, edge-coloring $G$ with $\Delta + \sqrt{\Delta}$ colors requires $O(\Delta^{2.25} \log^3 \Delta \log n + \Delta^2 \log^4 n)$ time using $O(n^2 \sqrt{\Delta} + n \Delta^2)$ processors on a COMMON CRCW PRAM.

**Proof.** Let $t = 1/2 \left\lfloor \log \Delta \right\rfloor - 1$, $t' = [1/2 \log \Delta - \log 3]$, and let $s = \max\{t, t'\}$. By Theorem 9.5, the corollary then follows. □

**Corollary 9.2** Given a graph $G(V,E)$ with maximum vertex degree $\Delta$, edge-coloring $G$ with $2.5 \Delta$ colors requires $O(\log \Delta \log n)$ time and $O(m + n)$ processors on a CREW PRAM.
Proof. By assigning $s = k$ in Theorem 9.5, the maximum degree of any subgraph in the final partition is bounded by 3. Color each subgraph by applying the algorithm in [96], which requires $O(\log n)$ time using $O(n)$ processors, instead of applying Algorithm 3. The total number of colors used is $\lambda(k - 1, k) + 3 \cdot 2^{k-1} \leq \Delta + 2^k + 1/2 \cdot 2^k \leq 2.5\Delta$. \qed

9.6 Conclusions

In this chapter, we studied the edge-coloring update problem, the edge-coloring problem, and the approximate edge-coloring problem in the parallel context, and presented fast parallel algorithms for them. The proposed parallel algorithms have the best time complexities so far.
Chapter 10

An NC Algorithm for Computing Maximal Interlocking Sets

10.1 Introduction

Consider a family of intervals $\mathcal{I} = \{I_i \mid I_i = [a_i, b_i] \text{ and } 1 \leq i \leq n\}$ on the real line where $a_i$ and $b_i$ are the left and the right endpoints of $I_i$ respectively. We say that two intervals $[a_i, b_i]$ and $[a_j, b_j]$ are interlocked if either $a_i < a_j < b_j < b_i$ or $a_j < a_i < b_i < b_j$. An interlocking set is a set of intervals in which every two intervals are related in the symmetric-transitive closure of the interlock relation. The problem concerned here is to partition $\mathcal{I}$ into maximal interlocking sets. It should be mentioned that this problem has practical application in VLSI layout. For example, the determination of interlocking sets is used in single-row routings [145, 153]. There, a set of two-point nets lying in a single row is given, the goal is to find a routing of these nets that minimizes the maximum channel width provided the routing exists. The solutions to this routing problem are given in [145, 153]. A key element in the solutions is partitioning of the nets into maximal interlocking sets.

A straightforward method to solve the problem is to construct an interlock graph $G = (V, E)$ having one vertex for each interval and an edge between vertices $v$ and $u$ if and only if intervals $v$ and $u$ are interlocked. The maximal interlocking sets in $\mathcal{I}$ then corresponds to the connected components (CCs for short) of $G$. Obviously there is a trivial approach to construct $G$ in $O(n^2)$ time. Then computing the CCs of $G$ takes $O(n^2)$ time because $|E| = \Omega(n^2)$ in the worst case. In this chapter we show it suffices to compute the CCs of graph $G_1 = (V, E')$ instead of computing the CCs of $G$, where $G_1$ is a spanning subgraph of $G$ and $|E'| \leq 2n - 2$. We later show that the CCs of $G_1$ are the CCs of $G$ exactly.

There were some studies regarding this problem, Raghavan and Shani [145] presented an $O(n^2)$ time algorithm using dynamic programming technique. After investigating the property of this special graph, Lloyd [127] presented an $O(n\alpha(2n, n))$ time sequential algorithm provided the endpoints of intervals are sorted already. However, his algorithm seems highly sequential. Later, Saxena and Prasad [153] studied this problem again, and presented a parallel algorithm which requires either $O(\log^2 n)$ time and $O(n)$ processors on a CREW PRAM, or $O(\log n \log \log n)$ time and $O(n)$ proces-
sors on an ARBITRARY CRCW PRAM. Meanwhile, they also gave an $O(n)$ time sequential algorithm provided the endpoints of intervals are integers, with the fact that $n$-integer sorting can be done in $O(n)$ time. Otherwise, their sequential algorithm still needs $O(n \log n)$ time.

In this chapter we present an efficient parallel algorithm for the problem. Our algorithm requires either $O((\log n \log \log n)$ time and $O(n/\log \log n)$ processors on an EREW PRAM, or $O(\log n)$ time and $O(n)$ processors on an ARBITRARY CRCW PRAM. Compared with the best known algorithm currently — Saxena and Prasad’s algorithm [153], we improve by either an $O((\log n/\log \log n)$ factor in time using fewer processors on an EREW PRAM—a weaker model than the CREW PRAM, or an $O((\log \log n)$ factor in time using the same number of processors on the same CRCW PRAM. When the endpoints of intervals are sorted, the sequential implementation of our algorithm needs $O(n)$ time only, improving Lloyd’s sequential algorithm by an $O(2n,n)$ factor.

The rest of this chapter is organized as follows. In Section 10.2 we introduce some concepts and lemmas. The algorithm and its correctness proof are shown in Section 10.3. A conclusion is given in Section 10.4.

## 10.2 Preliminaries

Before we proceed, we present some related results to our problem that will be used later.

The all nearest larger values problem (ANLV for short) can be described as follows: given a real number sequence $r_1, r_2, \ldots, r_n$, for each element $r_i$, determine the nearest element of its left and the nearest element of its right that are larger than it, i.e., for every $i$, find the largest $j$ and the smallest $k$ such that $r_i < r_j$ and $r_i < r_k$ where $1 \leq i \leq n$, $1 \leq j < i$, and $i < k \leq n$. We denote $r_j$ and $r_k$ by $ANLV_L(r_i)$ and $ANLV_R(r_i)$ respectively. If there does not exist such a $r_j$ or $r_k$, we just set $ANLV_L(r_i) = \emptyset$ or $ANLV_R(r_i) = \emptyset$. Kim [105] already showed that the ANLV problem can be solved in the following time and processor bounds.

**Lemma 10.1** [105] The ANLV problem can be solved in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

Similarly, the all nearest smaller values problem (ANSV for short) can be described as follows: given a real number sequence $r_1, r_2, \ldots, r_n$, for each element $r_i$, determine the nearest element of its left and the nearest element of its right that are smaller than it, i.e., for every $i$, find the largest $j$ and the smallest $k$ such that $r_i > r_j$ and $r_i > r_k$ where $1 \leq i \leq n$, $1 \leq j < i$, and $i < k \leq n$. We denote $r_j$ and $r_k$ by $ANSV_L(r_i)$ and $ANSV_R(r_i)$ respectively. If there does not exist such a $r_j$ or $r_k$, we just set $ANSV_L(r_i) = \emptyset$ or $ANSV_R(r_i) = \emptyset$.

**Lemma 10.2** The ANSV problem can be solved in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.
Proof. We prove this lemma by reducing the ANSV problem to the ANLV problem. Let \( R = r_1, r_2, \ldots, r_n \) be the original sequence. We obtain another sequence \( R' = -r_1, -r_2, \ldots, -r_n \). It is straightforward to show that the ANLV problem on \( R' \) is equal to the ANSV problem on \( R \). While \( R' \) can be produced in \( O(\log n) \) time using \( O(n/\log n) \) processors on an EREW PRAM. By Lemma 10.1, the lemma follows. \( \Box \)

10.3 The Algorithm

10.3.1 The algorithmic description

For each interval \( I_i = [a_i, b_i] \), denote by \( |I_i| = |a_i - b_i| \) the length of \( I_i \).
Let \( L_{left}(i) = \{ I_j \mid |I_i \cap I_j| = \max_{j'} \{|I_i \cap I_{j'}| \mid a_{j'} < a_i < b_{j'} < b_i\}\} \). Define the nearest left neighboring interval, denoted by \( L(I_i) \), of \( I_i \) as follows.

\[
L(I_i) = \begin{cases} 
I_j, & \text{if } a_j = \min \{ a_{j'} \mid I_{j'} \in L_{left}(i) \text{ and } I_{j'} = [a_{j'}, b_{j'}] \} \\
\emptyset, & \text{otherwise}
\end{cases}
\]

Similarly, let \( R_{right}(i) = \{ I_j \mid |I_i \cap I_j| = \max_{j'} \{|I_i \cap I_{j'}| \mid a_{j'} < a_i < b_{j'} < b_i\}\} \). Define the nearest right neighboring interval, denoted by \( R(I_i) \), of \( I_i \) as follows.

\[
R(I_i) = \begin{cases} 
I_j, & \text{if } b_j = \max \{ b_{j'} \mid I_{j'} \in R_{right}(i) \text{ and } I_{j'} = [a_{j'}, b_{j'}] \} \\
\emptyset, & \text{otherwise}
\end{cases}
\]

The problem is to compute all maximal interlocking sets of \( I \), i.e., compute all CCs of graph \( G(V, E) \). The basic idea behind our algorithm is that we compute all CCs of a subgraph \( G_1 = (V, E') \) of \( G \) instead of \( G \) itself, where \( E' = \{ (I_i, L(I_i)) \mid 1 \leq i \leq n \} \cup \{ (I_i, R(I_i)) \mid 1 \leq i \leq n \} \). Obviously \( E' \subseteq E \) and \( |E'| \leq 2n - 2 \). We later show that the CCs of \( G_1 \) are the CCs of \( G \) exactly. Basically, the algorithm for computing all CCs of \( G \) consists of the following steps.

Step 1. Compute \( R(I_i) \) for every interval \( I_i \in I \);
Step 2. Compute \( L(I_i) \) for every interval \( I_i \in I \);
Step 3. Represent the graph \( G_1 \) by the adjacency lists of its vertices;
Step 4. Compute all CCs of \( G_1 \), i.e., all CCs of \( G \).

In the following we present the implementation details of each step above.

Step 1. We first sort the family of intervals \( I \) in ascending order according to the left endpoints of intervals. Let \( I_1, I_2, \ldots, I_n \) be the original sequence of intervals, \( A_{init} = a_1, a_2, \ldots, a_n \) be its corresponding left endpoint sequence, and \( B_{init} = b_1, b_2, \ldots, b_n \) be its corresponding right endpoint sequence. We sort \( A_{init} \) by increasing order. Let \( a_1, a_2, \ldots, a_n \) be the sorted interval sequence, where \( a_j \leq a_k \) for all \( j \) and \( k, 1 \leq j < k \leq n \). During sorting, if \( a_i = a_j \), then the ranks of \( a_i \) and \( a_j \) are determined by the following procedure. We compare \( b_i \) and \( b_j \), if \( b_i > b_j \) then the rank of \( a_i \) is smaller than
the rank of \( a_j \). Meanwhile, we also exchange the positions of \( b_i \) and \( b_j \) in \( B_{\text{init}} \). Let \( B = b_1, b_2, \ldots, b_n \) be the final sequence of \( B_{\text{init}} \). We then solve the ANLV problem on \( B \). As a result, we obtain \( \text{ANLV}_L(b_i) \) and \( \text{ANLV}_R(b_i) \), \( 1 \leq j \leq n \). Assume that the two endpoints of an interval know the index of the interval. For example, the endpoint \( b_j \) knows that its interval index is \( i_j \). If \( \text{ANLV}_R(b_i) \neq \emptyset \) and \( \text{ANLV}_R(b_i) = b_{i_k} \), then \( j < k \) obviously. We thus have the following lemma.

**Lemma 10.3** If \( \text{ANLV}_R(b_i) \neq \emptyset \) and \( \text{ANLV}_R(b_i) = b_{i_k} \), then

\[
R(I_{i_j}) = \begin{cases} 
I_{i_k}, & \text{if } \text{ANLV}_R(b_i) = b_{i_k} \text{ and } a_{i_k} \leq b_{i_j} \\
\emptyset, & \text{otherwise}
\end{cases}
\]

**Proof.** We show that if \( \text{ANLV}_R(b_i) \neq \emptyset \) and \( \text{ANLV}_R(b_i) = b_{i_k} \), \( R(I_{i_j}) = I_{i_k} \) provided \( a_{i_k} \leq b_{i_j} \). By the definition of \( R(I_{i_j}) \), it is impossible that an interval \( I_{i_{k'}} \) with \( k' < j \) is a candidate of \( R(I_{i_j}) \) because \( a_{i_{k'}} \leq a_{i_j} \). Now consider an interval \( I_{i_{k'}} \) with \( i < k'' < k \), it is also impossible that this interval is a candidate of \( R(I_{i_j}) \) because \( b_k \) is the first element at the right side of \( b_{i_j} \) in \( B \) which is larger than \( b_{i_j} \), i.e., all elements \( b_{i_{k'}} \leq b_{i_j} \). Meanwhile we also know \( a_{i_j} \leq b_{i_{k''}} \). By the definition of interlocking sets, there is no any edge in \( G \) connected the vertex \( I_{i_j} \) and the vertex \( I_{i_k} \), therefore, it is impossible that \( I_{i_{k''}} \) is a candidate of \( R(I_{i_j}) \). The rest is to show that \( I_{i_k} \) is the unique candidate of \( R(I_{i_j}) \) if \( a_{i_k} \leq b_{i_j} \). Assume that \( R(I_{i_j}) = I_{i_s} \) with \( k < s \), we have

\[
|I_{i_j} \cap I_{i_k}| = |a_{i_k} - b_{i_j}|
\]

\[
|I_{i_j} \cap I_{i_s}| = |a_{i_s} - b_{i_j}|
\]

Since \( a_{i_k} \leq a_{i_s} \),

\[
|I_{i_j} \cap I_{i_k}| \geq |I_{i_j} \cap I_{i_s}|
\]

which contradicts the definition of \( R(I_{i_j}) \). \( \square \)

By Lemmas 10.1 and 10.2, Step 1 can be done in \( O(\log n) \) time using \( O(n/\log n) \) processors on an EREW PRAM provided the endpoints of intervals are sorted. Otherwise the sorting on an EREW PRAM requires \( O(\log n) \) time and \( O(n) \) processors [30]. By Brent’s Theorem, sorting can also be finished in \( O(\log n \log \log n) \) time using \( O(n/\log \log n) \) processors on the same model.

Step 2. The computation of \( L(I_i) \) for each \( I_i \in I \) is similar to that used for \( R(I_i) \) in Step 1. Let \( A_{\text{init}} \) and \( B_{\text{init}} \) be defined as above. We first sort \( B_{\text{init}} \) by ascending order. During sorting, if \( b_i = b_j \), then the ranks of \( b_i \) and \( b_j \) are determined by the following procedure. We compare \( a_i \) and \( a_j \), if \( a_i > a_j \) then the rank of \( b_i \) is smaller than the rank of \( b_j \). Meanwhile, we also exchange the positions of \( a_i \) and \( a_j \) in \( A_{\text{init}} \). Let \( A = a_1, a_2, \ldots, a_n \) be the final sequence of \( A_{\text{init}} \). We then solve the ANSV problem on \( A \). As a result, all \( \text{ANSV}_L(a_{i_j}) \) and \( \text{ANSV}_R(a_{i_j}) \) are given, \( 1 \leq i \leq n \). If \( \text{ANSV}_L(a_{i_j}) \neq \emptyset \), let \( \text{ANSV}_L(a_{i_j}) = a_{j_k} \). Obviously \( i > k \). We thus have the following lemma.
Lemma 10.4 If AN SVL(a_jk) ≠ ∅ and AN SVL(a_ji) = a_i, then

\[ L(I_{jk}) = \begin{cases} I_{jk}, & \text{if } AN SVL(a_{ji}) = a_{jk} \text{ and } a_{ji} \leq b_{jk} \\ \emptyset, & \text{otherwise} \end{cases} \]

Proof. The proof of this lemma is similar to the one used in Lemma 10.1, and is omitted. □

Step 3. The construction of \( G_1(V, E') \) is quite easy. In the following we describe how to construct it in both sequential and parallel environments respectively. In the sequential environment, initially for each vertex \( v \), there are two pointers which point the head and the rear of the adjacency list of \( v \). When we put the vertex \( L(I_i) (R(I_i)) \) into the rear of the adjacency list of \( I_i \), we also put \( I_i \) into the rear of the adjacency list of \( L(I_i) (R(I_i)) \). So, the construction of \( G_1 \) can be done in \( O(n) \) time. Note that a vertex may appear in an adjacency list more than once, but this does not have any influence on the computation of CCs of \( G_1 \). In the parallel environment, we first sort all edges in \( E' \) by the lexical order of their endpoints, then use prefix computation to decide which vertices belong to the adjacency list of a certain vertex. The entire processing needs \( O(\log n) \) time using \( O(n) \) processors because the dominant computational complexity is sorting which requires \( O(\log n) \) time and \( O(n) \) processors on an EREW PRAM [30].

Step 4. Having \( G_1 \), the computation of CCs of \( G_1 \) is trivial. In the sequential environment, we use the depth-first search technique to find all CCs of \( G_1 \). This costs \( O(n) \) time because \( |E'| \leq 2n - 2 \) [2]. In the parallel environment, we use either the parallel algorithm of computing CCs due to Chong and Lam [24] which requires \( O(\log n \log \log n) \) time and \( O(n/ \log \log n) \) processors on an EREW PRAM, or the parallel algorithm by Cole and Vishkin [31] which requires \( O(\log n) \) time and \( O(n \alpha(2n, n)/ \log n) \) processors on an ARBITRARY CRCW PRAM.

In summary, we have the following theorem.

**Theorem 10.1** Given a family of intervals \( \mathcal{I} \), let \( G = (V, E) \) be the interlock graph of \( \mathcal{I} \). Then all CCs of \( G \) can be found either in \( O(\log n \log \log n) \) time using \( O(n/ \log \log n) \) processors on an EREW PRAM, or in \( O(\log n) \) time using \( O(n) \) processors on an ARBITRARY CRCW PRAM. Particularly, the computation of CCs of \( G \) can be done in \( O(n) \) sequential time when the endpoints of the intervals are sorted.

10.3.2 The correctness proof

The algorithm is quite simple. However, its correctness proof seems not easy. Here we show that the proposed algorithm is correct by giving the following theorem.

**Theorem 10.2** The CCs of \( G_1 \) are the CCs of \( G \) exactly.

Proof. We prove it by induction on the number \( k \) of intervals. Initially when \( k = 2 \), the theorem obviously holds. Assume that when \( k \leq n - 1 \), the theorem holds. Now we
show that it still holds when \( k = n \). Recall that \( G \) is the interlock graph of \( \mathcal{I} \), and \( G_1 \) is the subgraph of \( G \) defined in Section 10.3.1. Let the smallest left endpoint of intervals in \( \mathcal{I} \) be \( a \), define \( \mathcal{F} = \{ I_i \mid a_i = a \text{ and } I_i \in \mathcal{I} \} \). We select an interval \( I = [a, b] \) from \( \mathcal{F} \) such that \( I \) has the largest right endpoint. Let \( G' \) be the interlock graph formed by the intervals in \( \mathcal{I} - \{ I \} \). By the induction assumption, the theorem holds for graph \( G' \). Now we need to show that the theorem also holds for graph \( G \). In order to do that, it suffices to show the following two cases. Case (i) if an interval \( I_i \in \mathcal{I} - \{ I \} \) and \( I \) are interlocked, then by the definition of the interlock graph \( I_i \) and \( I \) are in the same CC of \( G \). We will show that \( I \) and \( I_i \) are in the same CC of \( G_1 \) too. Case (ii) if \( I_i = [a_i, b_i] \) and \( I_j = [a_j, b_j] \) are in the same CC of \( G' \), then they are in the same CC of \( G \). We will show that \( I_i \) and \( I_j \) are in the same CC of \( G_1 \) also.

Before we proceed, we define \( L^0(I_i) = I_i \) and \( L^k(I_i) = L(L^{k-1}(I_i)) \) if both \( L^k(I_i) \) and \( L^{k-1}(I_i) \) exist. Similarly \( R^k(I_i) = R(R^{k-1}(I_i)) \) can be defined.

We start by Case (i). We look at the interval sequence \( I_1, L(I_1), L^2(I_1), \ldots, L^c(I_1) \) where \( L^{c+1}(I_1) = \emptyset \) and \( c \) is a positive integer between 0 and \( n - 1 \). We claim that \( L^c(I_i) = I \). If this claim is true, then \( I \) and \( I_i \) are in the same CC of \( G_1 \) because \( \{(L^j(I_i), L^{j+1}(I_i)) \mid 0 \leq j \leq c - 1\} \subseteq E' \). Otherwise assume that \( L^c(I_i) \neq I \). Let \( L^c(I_i) \) be the last interval in the above sequence with its right endpoint \( b' > b \) where \( c' < c \). Then by the definition of the nearest left neighboring interval, \( L(L^{c'}(I_i)) = I \neq L^{c'+1}(I_i) \) which contradicts our initial assumption that \( I \) does not appear in the sequence.

The proof of Case (ii) is as follows. Let \( I_i \) and \( I_j \) be in the same CC of \( G' \). Let \( G'' = (V, E'') \) be a subgraph of \( G' \) and \( E'' = E_L \cup E_R \) where \( E_L = \{(I_i, L(I_i)) \mid I_i \in \mathcal{I} - \{ I \}\} \) and \( E_R = \{(I_i, R(I_i)) \mid I_i \in \mathcal{I} - \{ I \}\} \). By the induction assumption, \( I_i \) and \( I_j \) are in the same CC of \( G'' \). We notice that if \( I_i \) and \( I_j \) are in the same CC of \( G' \) by only the edges in \( E_R \), they are still in the same CC of \( G_1 \) because adding \( I \) into \( \mathcal{I} - \{ I \} \) does not change the nearest right neighboring interval of every interval in \( \mathcal{I} - \{ I \} \). So, the addition of \( I \) into \( \mathcal{I} - \{ I \} \) only involves changing the nearest left neighboring interval relation, i.e., the edge set \( E_L \). Let \( E'_L \subseteq E_L \) be a smallest edge subset such that \( I_i \) and \( I_j \) are in the same CC of \( G'' \). If adding \( I \) to \( \mathcal{I} - \{ I \} \) does not make any update to \( E'_L \), then \( I_i \) and \( I_j \) are still in the same CC of \( G_1 \) because \( E'_L \cup E_R \subseteq E' \). Otherwise assume that at least an edge \( (I_p, I_q) \in E'_L \) is updated due to adding \( I \), i.e., \( (I_p, I_q) \notin E' \). Our goal is to prove that \( I_p \) and \( I_q \) are still in the same CC of \( G_1 \). In order to do that, we show that either \( I_p \) and \( I_q \) are connected by the edges in \( E_R \); or \( I_p \) and \( I_q \) are in the same CC and \( I \) and \( I_p \) are in the same CC of \( G_1 \), therefore \( I_p \) and \( I_q \) are in the same CC of \( G_1 \), \( I_i \) and \( I_j \) are in the same CC of \( G_1 \). Without loss of generality, assume that \( a_p < a_q \). We have the following claims for \( G_1 \). (1) \( L(I_q) = I; \) and (2) \( I_p \subseteq I \).

The proof of Claim (1) is obvious because the endpoints of all the other intervals are not changed. By this claim, we know that \( I \) and \( I_q \) are in the same CC of \( G_1 \) because \( (I_q, L(I_q)) \in E' \) and \( L(I_q) = I \).

We now show Claim (2). Since \( a_p < a_q \) and \( b_p < b_q \) \((I_p \) and \( I_q \) are interlocked initially), by Claim (1), we derive that \( b < b_q \) and \( b_p < b \). Furthermore, by the assumption of \( I \), it is easy to derive that \( a \leq a_p < b_q \). Therefore, \( a \leq a_p < b_p \leq b \). The remaining goal is to show that either \( I_p \) and \( I_q \) are connected by the edges of \( E_R \), or \( I \)
and $I_p$ are in the same CC of $G_1$. Consider the sequence $I_p, R(I_p), R^2(I_p), \ldots, R^l(I_p)$ where $R^{l+1}(I_p) = \emptyset$. If $I_q$ appears in this sequence, then $I_p$ and $I_q$ are in the same CC of $G_1$, therefore $I_i$ and $I_j$ are in the same CC of $G_1$, the theorem follows. Otherwise, by the assumption that $I_p$ and $I_q$ are interlocked, there must exist an interval $I_s = [a_s, b_s]$ in the sequence such that $a_p < a_s \leq a_q$ and $b_s \geq b_q$ and the rank of $I_s$ in the sequence is the maximum, i.e., $I_q \subseteq I_s$, see Fig. 10.1. Clearly $I_p$ and $I_s$ are in the same CC of $G_1$. It is also trivial to show that $I_s$ and $I$ are interlocked because $a < a_s$ and $b_s \geq b_q > b$. By the discussion in Case (i), we know that $I$ and $I_s$ are in the same CC of $G_1$. Combining the above results, we know that $I_q$ and $I$ are in the same CC of $G_1$ by Claim (1), and $I_p$ and $I$ are in the same CC of $G_1$. Therefore, $I_p$ and $I_q$ are in the same CC of $G_1$. Thus, $I_i$ and $I_j$ are in the same CC of $G_1$. \(\square\)

### 10.4 Conclusion

In this chapter an efficient parallel algorithm for finding all maximal interlocking sets in a family of intervals has been presented. Our parallel algorithm improves the previous best results for this problem in a variety of parallel computational models.
Conclusions and Open Problems

In this thesis, we have investigated the parallel solutions for those graph problems that share at least one of the following features: the problems involve dynamic updates in some sense; the sparsification technique is applicable; or the problems are related to communications networks. The proposed parallel solutions combine a variety of graph properties and novel algorithmic techniques.

First we described the first NC algorithms for the fully dynamic maintenance of $k$-connectivity of a graph with $k = 2,3$ using $O(n)$ processors only. In particular the algorithm for the fully dynamic maintenance of 2ECCs requires $O(\log n \log \frac{n}{\log n})$ time and $O(n^{3/4})$ processors only. However, there still exists a gap between the sequential time complexity and the amount of work needed in parallel for this problem. So, here we have an open problem, that is, whether there exists a better NC algorithm for the problem without using more than $o(n)$ processors to narrow the gap, for example, is it possible that there is an NC algorithm for the 2ECC problem with only $O(\sqrt{n})$ processors? To answer this problem might be a challenging task which deserves to be further investigated. As for the fully dynamic maintenance of biconnectivity, is it possible to design an NC algorithm with $o(n)$ processors only? The answer for this problem may be much harder than that for the first one.

Next we devised the first NC algorithm for maintaining a solution to the all pairs shortest paths problem under edge insertions on a directed weighted graph. Our NC algorithm needs $O(n^2)$ processors only. We also generalized this result to longest paths in a DAG, topological sorting in a DAG, and transitive closure of a directed graph. But consider the edge deletion only, whether the dynamic maintenance of a solution of the all pairs shortest paths problem can be done in $O(\log^2 n)$ time using $O(n^2)$ processors or $O(nm)$ processors is still an open problem for constant $c$. Here the most outstanding question is whether or not there is an NC algorithm with $O(n^2)$ processors for the fully dynamic maintenance to a solution of the all pairs shortest paths problem under both edge insertion and edge deletion operations.

Third, we have presented an NC algorithm for testing $k$-connectivity on a directed graph with fixed $k$. We have shown testing $k$-connectivity on a directed graph with fixed $k$ is in NC. It might be interesting to investigate whether or not testing for $k$-connectivity with arbitrary $k$ on a directed/undirected graph is NC.

Fourth, we also studied the $k$-connectivity augmentation problem with a given
feasible set of weighted edges. We presented the first NC approximation algorithms with \(\Omega(\log n)\) performance ratio for 2-connectivity augmentation and \(k\)-edge connectivity augmentation with fixed \(k > 2\). It should be mentioned that there seems little hope of deriving an NC approximation algorithm with constant performance ratio for the above problems from previously known sequential algorithms. So, whether or not there is an NC approximation algorithm with constant performance ratio for each of them is still an outstanding open problem. We also noted that the performance ratio of our NC approximation algorithm for biconnectivity is \(O(\log^2 n)\), so one possible research topic for the problem is to explore whether there is an NC approximation algorithm with \(O(\log n)\) performance ratio.

Fifth, we dealt with \(k\)-connectivity on the weighted and unweighted graphs in the previous several chapters. In Chapter 6 we took into account the routing issue in communications networks, i.e., the \(k\) MVE MST problem. For a fixed \(k\) we presented the best result for the problem so far in the sequential and parallel environments. However, with arbitrary \(k\), until now we only knew that there existed an approximation algorithm with \(\Omega(\log k)\) performance ratio for the problem. One of the outstanding open problems is whether or not there is an NC approximation algorithm for the problem with constant performance ratio. We also investigated the implementing NC algorithms for the 1 MVE MST problem and the 1 MVE SP problem on meshes and hypercubes. The algorithm for the 1 MVE MST problem on the hypercube array is inefficient in the sense of amount of work taken, this leaves us an open problem. That is, whether there exists a better algorithm for this problem on this model which requires \(O(\log^2 n)\) time using \(n^2\) rather than \(n^3\) processors only.

Sixth, the MST of a graph is not usually the best candidate used as the high-level routing protocol in a communications network. Instead, the \(t\)-spanner of the graph is often used as this kind of protocol. So, a fast parallel algorithm for finding \(t\)-spanners is desired. In this thesis we devised parallel algorithms for finding approximate \(t\)-spanners on both weighted and unweighted graphs. However, whether or not there exists an NC algorithm for the problem is still unknown. To give a positive or negative answer for it is worthwhile investigating in the future.

Seventh, the edge-coloring problem in its own right is an interesting problem. In this thesis we presented an improved parallel algorithm for it. The proposed parallel algorithm has the best time complexity so far. However, whether or not it is in NC is still an outstanding open problem which deserves to be investigated further. We even relax the conditions by allowing \(\Delta + \sqrt{\Delta}\) colors rather than \(\Delta + 1\) colors to edge-color a graph \(G\); whether or not this approximate edge-coloring is in NC is also open.

Lastly, we studied the maximal interlocking set problem in a collection of intervals by giving a fast NC algorithm. But for this special problem, whether we can further reduce its time complexity on an EREW PRAM is still open. For example, whether or not there is such an NC algorithm with \(O(\log n)\) time using \(O(n)\) processors on an EREW PRAM is open.
Bibliography


