GRAPH AUGMENTATION AND RELATED PROBLEMS: THEORY AND PRACTICE

by

TSAN-SHENG HSU, B.S., M.S.C.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1993

In memory of my grandmother Yuan Hsu (1912 – 1991)

Acknowledgments

I would like to express my gratitude to my advisor, Professor Vijaya Ramachandran, for her guidance and support throughout the course of this work. Her numerous suggestions and comments were particularly beneficial. I am also indebted to all the other members of my dissertation committee, Professors Robert Boyer, David Matula, Greg Plaxton, and Martin D. F. Wong, for their valuable comments.

At different stages of this thesis, I have benefited from many useful discussions with Sanjoy Baruah, Jianer Chen, Tim Collins, Brandon Dixen, Harold Gabow, Tibor Jordán, Goos Kant, Arkady Kanevsky, G. Neelakantan Kartha, Pierre Kelsen, Dian Lopez, Don Padgett, Robert Read, Nandit Soparkar, Torsten Suel, Roberto Tamassia, and Honghua Yang. During the years in Austin I have also benefited from the help of many friends. I am grateful to all of them.

Most of the work reported in this thesis is joint work with Vijaya Ramachandran. Most of the work reported in Part II for implementing parallel graph algorithms is also joint work with Nathaniel Dean. I would like to thank Tibor Jordán, Arkady Kanevsky and Roberto Tamassia for providing useful information in developing Chapter 6, Baruch Schieber for helpful comments on an earlier version of Chapter 7, Monika Mevenkamp for her help in developing the interface between the parallel programs on the MasPar and NETPAD, Peter Winkler for helpful discussions on how to generate two-edge-connected graphs, Clyde Monma for his support of the implementation project reported in Part

II, and Michael B. Carter for providing a set of routines for measuring CPU time used by programs run on the MasPar.

This work has been supported in part by an MCD Fellowship of the University of Texas at Austin, by the National Science Foundation under Grants CCR-89-10707 and CCR-90-23059, by the Texas Advance Research Program under Grant 003658480, and by an IBM Graduate Fellowship. Part of the work reported in Chapter 11 was done while I was visiting Bellcore.

Tsan-sheng Hsu

The University of Texas at Austin

December, 1993

GRAPH AUGMENTATION AND RELATED PROBLEMS: THEORY AND PRACTICE

Publication No. _____

Tsan-sheng Hsu, Ph.D.
The University of Texas at Austin, 1996

Supervisor: Vijaya Ramachandran

Graphs play an important role in modeling the underlying structure of many real-world problems. In this thesis, we investigate several interesting graph problems.

The first part of this thesis focuses on the problem of finding a smallest set of edges whose addition makes an input undirected graph satisfy a given connectivity requirement. This is a fundamental graph-theoretic problem that has a wide variety of applications in designing reliable networks and database systems. We have obtained efficient sequential and parallel algorithms for finding smallest augmentations to satisfy 3-edge-connectivity, 4-edge-connectivity, biconnectivity, triconnectivity, and four-connectivity. Our parallel algorithms are developed on the PRAM model. Our approach in obtaining these results is to first construct a data structure that describes all essential information needed to augment the input graph, e.g., the set of all separating sets and the

set of all maximal subsets of vertices that are highly connected. Based on this data structure, we obtain a lower bound on the number of edges that need to be added and prove that this lower bound can be always reduced by one by properly adding an edge.

The second part of the thesis focuses on the implementation of PRAM-based efficient parallel graph algorithms on a massively parallel SIMD computer. This work was performed in two phases. In the first phase, we implemented a set of parallel graph algorithms with the constraint that the size of the input cannot be larger than the number of physical processors. For this, we first built a kernel which consists of commonly used routines. Then we implemented efficient parallel graph algorithms by calling routines in the kernel. In the second phase, we addressed and solved the issue of allocating virtual processors in our programs. Under our current implementation scheme, there is no bound on the number of virtual processors used in the programs as long as there is enough memory to store all the data required during the computation. The performance data obtained from extensive testing suggests that the extra overhead for simulating virtual processors is moderate and the performance of our code tracks theoretical predictions quite well.

Table of Contents

Chapter 1

Introduction

1.1 Background

Graph Augmentation

The problem of augmenting a graph to reach a given requirement (e.g., connectivity) by adding edges has important applications in designing networks with high reliability [FC70, SWK69], designing reliable database system [Esw73] and improving statistical data security [Gus87, Gus89]. The underlying model for solving these problems can be formulated as a graph. In the application, each addition of an edge incurs a certain cost. We refer to this problem as the *graph augmentation problem*.

A general formulation of the graph augmentation problem is as follows. Given an input graph G and a finite cost assigned to each pair of vertices in G, we want to add a set of edges with the minimum total cost such that the resulting graph G' satisfies a given property (e.g., high vertex-connectivity). The requirements for the input graph are called the *input requirements* and the requirements for the augmented graph are called the *output requirements*. Depending on the input requirements and the output requirements, we can map the graph augmentation problem to a variety of real-world problems. For example, in Chapters 4 and 5 we will present a linear-time sequential algorithm and an efficient parallel algorithm to triconnect an undirected graph using the smallest number of edges. This problem corresponds to the real-world problem of upgrading a (communication or computer interconnection) network such

that no failure of any two links or any two nodes can disconnect the whole network. The input graph represents the existing network that does not have the required connectivity. It is often the case in a local area network that the cost of adding a link is about the same among all possible links. The solution found by our algorithm given in Chapters 4 and 5 corresponds to the best way of upgrading the reliability of the existing network.

If the cost between each pair of vertices in the input graph is not uniform, we refer to this problem as the *minimum augmentation* problem. A set of edges with the minimum total cost whose addition meets the connectivity requirement is called a *minimum augmentation*. Finding a minimum augmentation can be very difficult for quite a few output requirements. For example, the decision problem associated with the minimum augmentation for reaching various connectivity requirements has been proven to be **NP-**complete. These results will be listed in Chapter 2.

The unweighted version of the graph augmentation problem is often easier than its weighted version. In the unweighted version of the problem, we want to augment the input graph to reach a given output requirement by adding a smallest set of edges. We refer to this problem as the *smallest augmentation* problem. A smallest set of edges whose addition meets the output requirement is called a *smallest augmentation*. We will show in Chapters 3 through 6 that the smallest augmentation problem can be solved efficiently for satisfying several vertex connectivity requirements.

We note that the complementary problem of deleting a maximum set of edges from a graph such that the resulting graph still satisfies certain properties has also been extensively studied for various connectivity requirements. See [HKRT92, KR91b, Kel92, KV92] for details.

Implementation of Efficient Parallel Graph Algorithms

In addition to the theoretical study of graph augmentation problems, we also study the implementation of graph algorithms. There has been a lot of work on implementing sequential graph algorithms [DMM92, MN89], and in this thesis, we will describe our implementation of some sequential graph algorithms. On the other hand, although several parallel graph algorithms have been developed [JáJ92, KR90, Lei92, Qui87], not much implementation has been done. In the second part of this thesis, we study the problem of implementing parallel graph algorithms efficiently on a massively parallel SIMD computer.

PRAM Models

In this thesis, we use the parallel random access machine (PRAM) as the model of parallel computation. A PRAM [KR90] consists of a set of random access machines (RAM) and a global memory. Each random access machine has a processor and a local memory. Each RAM has a unique ID numbered from 1 to the total number of processors in the PRAM machine. During each step of the computation, each processor synchronously executes the same instruction, but with possibly different operands. During each time cycle, a processor can read a global memory cell, perform some local computations on its local data and write data into a global memory cell. A schematic diagram of a PRAM is shown in Figure 1.1.

The PRAM model assumes that each instruction takes constant time no matter how many processors want to access the global memory. Depending on the type of global memory access allowed, the PRAM model can be further classified into the following models: EREW PRAM, CREW PRAM, CRCW PRAM, and ERCW PRAM. The EREW (exclusive-read exclusive-write) PRAM model requires that no two processors read or write the same

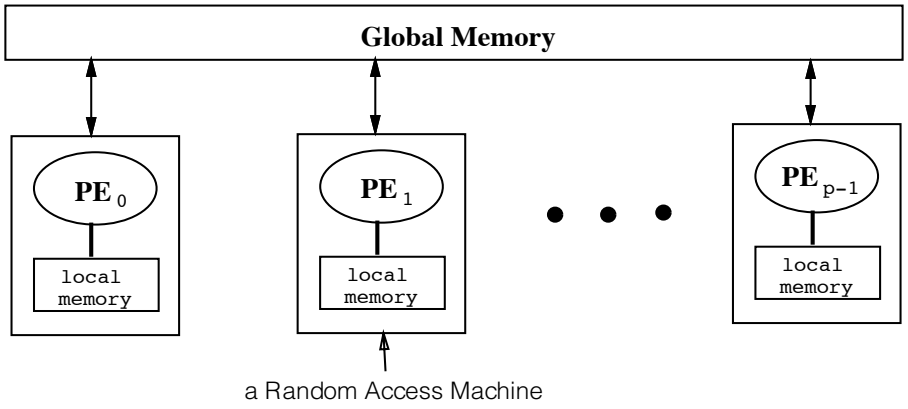


Figure 1.1: Schematic diagram for the PRAM model.

global memory location at any given time. The CREW (concurrent-read exclusive-write) PRAM model allows concurrent access to the same global memory location by more than one processor for reading, but does not allow more than one processor to write the same memory location at the same time. The CRCW (concurrent-read concurrent-write) PRAM models allow different processors to read and write into the same global memory location at the same time. In the case of writing different data into the same global memory location at a given time, we must define the result of the concurrent write. The COMMON CRCW model requires data that are written into a common global memory location by different processors to be the same at any given time. In the PRIORITY CRCW model, if several processors try to write different data in the same global memory location at any given time, the data sent by the processor with the least ID is written into the memory location. There are various other protocols for resolving collisions in writing. For details, see [KR90]. The ERCW (exclusive-read concurrent-write) PRAM models do not allow more than one processor to read the same memory location at the same time, but allow multiple processors to write into a memory location. The collision resolution protocols for concurrent write, discussed above for the CRCW models,

also apply here. Not too many results are known for the ERCW PRAM models [MR93].

Algorithmic Notation

The algorithmic notation used is pseudo-Pascal and is similar to the notation of Tarjan [Tar83] and Ramachandran [Ram93]. We enclose comments between '{∗' and '∗}'. Parameters are called by value unless they are declared with the keyword **modifies** in which case they are called by value and result. We use the following **pfor** statement for executing a loop in parallel.

pfor *iterator* **do** *statement list* **rofp**

The effect of this statement is to perform the statement list in parallel for each value of the iterator.

Throughout the thesis, let $\log x$ denote $\log_2 x$ for any variable x. We also use n to denote the number of vertices in the input graph and m to denote the number of edges in the input graph. In this thesis, a spanning forest of a graph G is a maximal subgraph of G (with respect to the edges in G) that is a forest.

1.2 Summary of Thesis Results

This thesis consists of two parts. The first part deals with the smallest augmentation problem when the output requirement is vertex connectivity. At the end of Part I, we describe our implementation of some smallest augmentation algorithms. The second part deals with the implementation of several efficient parallel graph algorithms.

Part I

We present the following results in Part I. Chapter 2 lists results known for solving the augmentation problem. Chapter 3 gives a corrected linear time

sequential algorithm and an efficient parallel algorithm for finding a smallest biconnectivity augmentation on an undirected graph. Our parallel algorithm runs in $O(\log^2 n)$ time using a linear number of processors on an EREW PRAM, where n is the number of vertices in the input graph.

Chapter 4 gives a linear time sequential algorithm and an efficient parallel algorithm for finding a smallest triconnectivity augmentation on a biconnected graph. Our parallel algorithm runs in $O(\log^2 n)$ time using a linear number of processors on an EREW PRAM. Building on the results in Chapter 4, Chapter 5 shows how to extend our augmentation algorithms (both the sequential algorithm and the parallel algorithm) to handle the case when the input graph is not biconnected. Algorithms given in Chapter 5 have the same complexities with the algorithms given in Chapter 4.

Chapter 6 gives an $O(n\alpha(m,n)+m)$ time sequential algorithm for finding a smallest augmentation to four-connect a triconnected graph with n vertices and m edges, where $\alpha(m,n)$ is the inverse Ackermann function.

By using an efficient method to transform the problem of finding a smallest edge-connectivity augmentation into the problem of finding a smallest vertex-connectivity augmentation, we give a linear time sequential algorithm and an efficient parallel algorithm for finding a smallest 3-edge-connectivity augmentation in Chapter 7. Our parallel algorithm runs in $O(\log n)$ time using $O(\frac{(n+m)\log\log n}{\log n})$ processors on a CRCW PRAM, where n is the number of vertices and m is the number of edges n the input graph. In Chapter 7, we also give an $O(n\alpha(m,n)+m)$ time sequential algorithm to 4-edge-connect an undirected graph using the smallest number of edges.

In Chapter 8, we describe our implementation of several smallest augmentation problems and provide performance data. Finally in Chapter 9, we summarize the results of Part I and list some open problems.

Part II

We present the following results in Part II. Chapter 10 gives an overview of the hardware and software of the massively parallel SIMD computer MasPar MP-1 and our general approach towards implementing parallel graph algorithms on the MP-1.

Chapter 11 shows the systematic approach we used to implement parallel graph algorithm when the input size is not larger than the number of available physical processors. Our approach was to first build a kernel which consists of commonly used routines. Then we implemented efficient parallel graph algorithms developed on the PRAM model by calling routines in the kernel. We also implemented the corresponding sequential graph algorithms. In addition to the description of our implementation, we provide including an analysis of the speed-up achieved.

Using a language that does not support the use of virtual processors, Chapter 12 describes techniques for implementing parallel programs with virtual processing. We first present our data allocation scheme for virtual processing and a set of translation rules for rewriting a code that uses no virtual processors into a code with virtual processing. We then describe the implementation and fine-tuning of a set of commonly used routines with virtual processing. In coding these routines, we tried different underlying algorithms. We present the performance data for our different implementations. We also present the performance data of our sequential code and compare it with that of our parallel code.

Using the techniques given in Chapter 12, we re-coded and fine-tuned our earlier parallel graph algorithms (described in Chapter 11) to incorporate the usage of virtual processors. The results are presented in Chapter 13. Under

the current implementation scheme, there is no limit on the number of virtual processors that one can use in the program as long as there is enough main memory to store all the data required during the computation. We also give two general optimization techniques to speed up our computation. The performance data we obtained from extensive testing suggests that the extra overhead for simulating virtual processors is moderate. Furthermore, the performance of our code tracks theoretical predictions quite well. In addition, our parallel code using virtual processing runs on much larger size inputs than our sequential code.

Finally in Chapter 14, we summerarize our work in Part II and list directions for future improvements.

Part I

Graph Augmentation Problems

Chapter 2

Survey of Known Results

In this chapter, we list results known for graph augmentation problems.

2.1 Augmenting Empty Graphs — Network Synthesis

The problem of how to design a network from scratch such that the network contains desirable properties is called the *network synthesis problem*. The corresponding graph augmentation problem requires that the input graph is empty. In the following subsections, we give some classical results and some recent results for augmenting empty graphs to satisfy various output requirements. The survey paper written by Christofides and Whitlock [CW81] describes results (obtained before 1981) on synthesizing networks with certain connectivity requirements.

2.1.1 Minimum Augmentation

In this subsection, we list some results about synthesizing networks when edges have non-uniform costs.

Monma, Munson, and Pulleyblank [MMP90] considered the problem of constructing a minimum-cost 2-edge-connected or biconnected graph whose cost function on edges satisfies the triangular inequality. They showed that there always exists a minimum-cost 2-edge-connected (biconnected) graph that satisfies the following conditions: (1) the degree of each its vertices is 2 or 3; (2)

the removal of any pair of edges (vertices) creates a cut edge (cutpoint) in one of the connected components of the resulting graph. They gave a polynomial-time approximation algorithm for constructing such a graph by finding a near-optimal Hamilton circuit whose total cost is at most $\frac{4}{3}$ times the total cost of an optimal 2-edge-connected (biconnected) graph.

Bienstock, Brickell, and Monma [BBM90] extended the result in [MMP90] to construct a minimum-cost k-edge-connected (k-vertex-connected) graph whose cost function on edges satisfies the triangular inequality. They first showed that there exists a minimum-cost k-edge-connected (k-vertex-connected) graph that satisfies the following conditions: (1) the degree of each of its vertices is k or $k+1$; (2) the removal of any w edges (vertices), $w \geq k$, creates a separating t-edge-set (vertex-set), $t \leq k$, in one of the connected components of the resulting graph. The number of vertices in the optimal k-vertex-connected graph is at least $2k$. For any fixed $k \geq 2$, they proposed a polynomial time approximation algorithm by finding a near-optimal Hamilton circuit and adding an edge between any pair of vertices that can reach one another by a path of $\leq \lceil \frac{k}{2} \rceil$ edges. By using the Christofides heuristic for finding a near-optimal Hamilton circuit, they can construct a graph whose cost is at most $3 \cdot k \cdot (k+1)$ times the optimal cost.

Grötschel and Monma [GM90] further extended results in [BBM90, MMP90] by associating the designing of an optimal k-edge-connected (k-vertex-connected) graph with the solution of a system of integer linear programs with properly defined constraints.

2.1.2 Smallest Augmentation

In this subsection, we list some results for synthesizing networks when the edges added have uniform cost.

Fulkerson and Shapley [FS71] gave an $O(kn^2)$ time algorithm to construct a minimum k-edge-connected undirected graph with n nodes, for an arbitrary given k. The graph they constructed has $n-1$ edges if $k=1$ and has $\lceil \frac{kn}{2} \rceil$ edges if $k>1$. Their algorithm included a linear time subroutine that given a minimum n-node k-edge-connected graph, generates a minimum k-edge-connected graph with $n+1$ nodes by changing (i.e., deleting and inserting) $O(k)$ edges.

For the case of reaching an arbitrary vertex-connectivity on an empty graph, Harary [Har62] constructed an n-node k-connected undirected graph with the smallest number of edges (the number is $n-1$ if $k=1$; otherwise, the number is $\lceil \frac{kn}{2} \rceil$) in $O(kn)$ time for an arbitrary k. Schumacher [Sch84] gave an $O(n^2)$ time algorithm for constructing an n-node k-connected undirected graph with the smallest number of edges, and a diameter which is no larger than twice the theoretical minimum. Both of their constructions are quite different from the method used by Fulkerson and Shapley [FS71] to construct their minimum n-node k-edge-connected undirected graphs.

Given d and k, Memmi and Raillard [MR82] constructed a regular graph with degree d and diameter k using the maximum number of vertices. Amar [Ama83] further proved that the graph constructed in [MR82] is d-vertex-connected. Imase, Soneoka, and Okada [ISO85] considered this problem on directed graphs.

2.2 Minimum Connectivity Augmentation

The problem of finding a minimum augmentation to connect an input graph is equivalent to the problem of finding a minimum-cost spanning tree. A survey of polynomial time sequential algorithms for this problem on undirected

graphs, directed graphs and for dealing with other additional constraints can be found in Tarjan [Tar83]. A history of this problem is described in Graham and Hell [GH85]. There are quite a few parallel algorithms for finding a minimum-cost spanning tree. For a survey of results, see Gibbons and Rytter [GR88], JáJá [JáJ92], Karp and Ramachandran [KR90], Quinn [Qui87], and the recent result of Chong and Lam [CL93].

2.2.1 NP-Completeness Results

The decision problem associated with the minimum augmentation problem is **NP**-complete for most of the connectivity requirements other than connectedness. (For the definition of **NP**-completeness and a survey of related results, see Garey and Johnson [GJ79].) Eswaran and Tarjan [ET76] proved that for achieving strong connectivity on a directed graph and for achieving 2-edge-connectivity or biconnectivity on an undirected graph, the minimum augmentation problem is **NP**-hard by a reduction from the Hamiltonian circuit problem. This version of the minimum augmentation problem remains **NP**-hard even if the input graph is connected. The problem of strongly connecting a *weakly connected* directed graph remains **NP**-hard. The reductions are made from the 3-dimensional matching problem [FJ81]. Watanabe and Nakamura [WN87] showed that the minimum augmentation problem for k-edge-connectivity or k-connectivity is **NP**-hard, for any $k \geq 2$. It is also shown in [WNN89] that for 3-edge-connectivity and triconnectivity, the minimum augmentation problem is **NP**-hard even if the input graph is *biconnected* by a reduction from the 3-dimensional matching problem.

Watanabe, Higashi, and Nakamura [WHN90] considered the following variation of the minimum augmentation problem. Given a graph G and a

subset of vertices S in G, we want to find a set of edges with the minimum cost whose addition makes every pair of vertices in S satisfy a given connectivity requirement. The addition of multiple copies of the same edge is not allowed. This problem is clearly **NP**-hard (even on undirected graphs) if one requires the resulting graph to have k vertex-disjoint or k edge-disjoint paths between every pair of vertices in S, for any $k \geq 2$. They showed that the problem is **NP**-hard if we are required to have a directed cycle between every pair of vertices in S on directed graphs, even if the values of the costs are all equal to 1.

2.2.2 Approximation Algorithms

Frederickson and JáJá [FJ81] gave approximation algorithms for directed graphs to achieve strong connectivity, and for undirected graphs to achieve 2-edge-connectivity and biconnectivity. Their algorithm ran in $O(n^2)$ time on an n-node graph where the sum of the costs of all edges added is at most twice the minimum cost when the input graph is connected. (See [GJ79, PS83] for a discussion on the definition of the performance ratios of approximation algorithms.) Khuller and Thurimella [KT92] gave an improved algorithm for approximating minimum biconnectivity augmentation within a factor of 2 in $O(m + n \log n)$ time when the input graph is connected. They also studied the problem of increasing the edge-connectivity of any graph to k using a set of edges whose cost is at most twice the optimal value. The gave an $O(kn \log n(m + n \log n))$ time algorithm for this problem. When k is odd, they also gave an $O(kn^2)$ time algorithm to increase the edge-connectivity from k to $k+1$, with a set of edges whose cost is within twice the optimal value. Watanabe and Nakamura [WN87] gave an $O(n^3)$ time approximation algorithm for achieving 3-edge-connectivity on an n-node 2-edge-connected graph G where

the sum of the costs of all edges added is no more than the sum of the costs of all the edges in a certain spanning tree of G; the resulting graph obtained from their algorithm is simple (that is, contains no multiple edges). Watanabe, Mashima, and Taoka [WMT92] gave several approximation algorithms for increasing the edge-connectivity of any graph by 1 in $O(n^3)$ time. They gave experimental data, but no theoretical analysis.

For approximation algorithms to increase connectivity within a given set of vertices, Watanabe, Higashi, and Nakamura [WHN90] gave an $O(n^2)$ time approximation algorithm for achieving 2-edge-connectivity on a specified set of vertices on undirected graphs. The sum of the costs of all edges added by the algorithm is no more than twice the minimum possible total cost if the input graph is connected. They also gave an $O(n^3)$ time approximation algorithm for achieving biconnectivity on a specified set of vertices on undirected graphs. The sum of the costs of all edges added by the algorithm is no more than four times the minimum possible if the input graph is connected. An $O(n^3)$ time approximation algorithm was also given for reaching strong connectivity on directed graphs; however, there is no performance ratio given on the total weight of the set of edges added by their algorithm.

Goemans and Williamson [GW92] solved the minimum augmentation problem when one requires the output graph to have at least one path between any vertex in Q_1 of p vertices and any vertex in Q_2, where Q_1 and Q_2 are given sets of vertices with cardinalities p. The approach they used is different from [WHN90] by encoding a set of constraints that can be solved using integer linear programming. They obtained a $2-\frac{1}{p}$ approximation result in $O(n^2 \log n)$ time. Using similar techniques, Ravi and Klein [RK93] gave a way to augment a graph such that there are at least two edge-disjoint paths between any pair

of vertices in a given set of p vertices. They obtained a $3 - \frac{3}{p}$ approximation result in $O(n^2 \log n)$ time. They could obtain a 2 approximation result with the same time bound if the given set of vertices is connected.

2.3 Smallest Edge-Connectivity Augmentation

For the problem of finding a smallest augmentation to reach a given edge-connectivity property, several polynomial time algorithms on undirected graphs, directed graphs and mixed graphs are known.

2.3.1 Undirected Graphs

Algorithms for finding a smallest augmentation to connect an undirected graph are well-known (that is, the problem of finding a spanning tree). We can first find connected components of the input graph by depth-first search, and then connect connected components by a spanning tree.

In [ET76], Eswaran and Tarjan gave a linear time algorithm for finding a smallest 2-edge-connectivity augmentation on undirected graphs by looking at the structure of the 2-edge-connected components and cut-edges. Watanabe, Narita, and Nakamura [WNN89] gave a polynomial time algorithm for achieving 3-edge-connectivity. Watanabe, Yamakado and Onaga [WYO91] gave an improved linear time algorithm for this problem. In Chapter 7 of this thesis, we give another linear time algorithm for reaching 3-edge-connectivity that can be parallelized. In that chapter, we also present an almost linear time algorithm for achieving 4-edge-connectivity.

For achieving arbitrary edge-connectivity, Ueno, Kajitani, and Wada [UKW88] described a polynomial time algorithm for solving this problem on a tree. No exact time bound was given. Watanabe [Wat87] and Watanabe

and Nakamura [WN87], gave the first polynomial time algorithms to solve the smallest augmentation problem for an arbitrary undirected graph to achieving a given edge-connectivity. Cai and Sun [CS89] also gave a polynomial time algorithm for solving the same problem.

Naor, Gusfield, and Martel [NGM90] gave an $O(\delta^2 nm + nF(n,m))$ time algorithm to increase the edge-connectivity of an undirected graph by δ, where n and m are the number of vertices and edges in G, respectively, and $F(n,m)$ is the time to perform one maximum flow on G. (The best known bound for $F(n,m)$ is $O(\min\{n^{\frac{2}{3}}m, m^{\frac{3}{2}}\})$ [ET75].) They first devised an $O(nm)$ time algorithm to optimally increase the edge-connectivity of an undirected graph by 1. By applying the basic algorithm δ times, they showed that by carefully choosing the edges added in each iteration, they could optimally increase the edge-connectivity of any graph by δ. Frank [Fra90, Fra92] solved a more general version of the smallest augmentation problem for reaching any arbitrary required edge-connectivity. Let a weight be assigned to each node in the input graph and let the cost of an edge be the sum of the weights of its two endpoints. He gave an $O(n^5)$ time algorithms for finding a set of edges with the minimum total weight whose addition makes the input graph (undirected or directed) to achieve any arbitrary required edge-connectivity. Gabow [Gab91] gave an improved algorithm for achieving τ-edge-connectivity on undirected graphs. His algorithms runs in $O(m + \tau^2 n \log n)$ time.

2.3.2 Directed Graphs

We describe results known for solving the smallest edge-connectivity augmentation problem on directed graphs in this section. Eswaran and Tarjan [ET76] gave a linear time algorithm for finding a set of edges whose addition

to strongly connect a directed graph by using the minimum number of edges. Their algorithm solved this problem by looking at all sinks and sources of the given directed graph.

Kajitani and Ueno [KU86] solved the problem of finding a smallest augmentation to k-edge-connect a directed tree in polynomial time (no exact time bound is given) for an arbitrary value k. Frank's results [Fra90, Fra92] mentioned in the previous section can also be used on directed graphs.

2.3.3 Mixed Graphs

For results on mixed graph, Gusfield [Gus87] considered the following smallest augmentation problem. Given a mixed graph, we want to add a smallest set of directed edges such that every pair of vertices in the resulting graph can reach one another. He gave a linear time algorithm in [Gus87] for solving the problem.

Note that all the above mentioned smallest augmentation algorithms [CS89, Fra92, KU86, UKW88, Wat87, WN87, WNN89] produce multi-graphs. We do not know how to solve the smallest augmentation problem in polynomial time for achieving an arbitrary edge-connectivity such that the resulting graph remains simple. We also do not know how to solve this problem if we just want to satisfy the given edge-connectivity requirement on a specified set of vertices.

2.4 Smallest Vertex-Connectivity Augmentation

The following results are known for solving the smallest augmentation problem on an undirected graph to satisfy a vertex-connectivity requirement.

Eswaran and Tarjan [ET76] (and Plesník [Ple76], independently) gave a lower bound for the smallest number of edges needed to biconnect an undi-

rected graph and proved that the lower bound can always be achieved. Rosenthal and Goldner [RG77] developed a linear time sequential algorithm for finding a smallest augmentation to biconnect a graph; however, the algorithm in [RG77] contains an error. Hsu and Ramachandran [HR91b] gave a corrected linear time sequential algorithm. An $O(\log^2 n)$ time parallel algorithm on an EREW PRAM using a linear number of processors for this problem was also given in Hsu and Ramachandran [HR91b]. We will describe the above two algorithms in Chapter 3.

Fernández-Baca and Williams [FBW89] considered the smallest augmentation problem for reaching 2-edge-connectivity, biconnectivity, and strong connectivity on hierarchically defined graphs. This version of the augmentation problem has applications in VLSI circuit design. They obtained polynomial time algorithms for the above problems.

Watanabe and Nakamura [WN93, WN88, WN90] gave an $O(n(n+m)^2)$ time sequential algorithm for finding a smallest augmentation to triconnect a graph with n vertices and m edges. Hsu and Ramachandran [HR91a] gave a linear time algorithm for this problem. (Independently, Jordán [Jor93b] gave a different linear time algorithm for the special case of optimally triconnecting a biconnected graph.) We will describe the above result in Chapters 4 and 5. For four-connecting a triconnected graph, Hsu [Hsu92] gave an $O(n\alpha(m,n)+m)$ time sequential algorithm, where $\alpha(m,n)$ is the inverse Ackermann function. This algorithm will be described in Chapter 6 of this thesis.

There is no polynomial time algorithm known for finding a smallest augmentation to k-vertex-connect an undirected graph, for $k > 4$. Although no polynomial time solution is known for this problem. Jordán [Jor93b] gave an approximation algorithm for undirected graphs that uses no more than $k-2$ edges to $(k+1)$-vertex-connect a k-vertex-connected graph.

A special variation of the smallest augmentation problem is to consider the problem of augmenting a planar graph such that the output graph is still planar and satisfies given connectivity properties. This version of the augmentation problem has applications in drawing planar graphs nicely on a plane [Kan93]. Kant and Bodlaender [KB91] showed that the problems of adding a smallest set of edges to reach biconnectivity (reducing from the 3 partition problem) and triconnectivity (reducing from the vertex cover problem) are **NP**-hard for planar graphs. They gave approximation algorithms for these two problems that add at most twice (biconnectivity) and $\frac{3}{2}$ (triconnectivity) the smallest number of edges. They also gave an algorithm for approximating 2-edge-connectivity to within a factor of 2. All of their algorithms run in $O(n \cdot \alpha(n,n) + m)$ time. The decision problem associated with the problem of finding a smallest 2-edge-connectivity augmentation on planar graphs is not known to be **NP**-complete. Kant and Bodlaender [KB92] also studied these problems while trying to minimize the maximum degree of the resulting graph, and Kant [Kan91] obtained some results for these problems for the case when the input graph is outerplanar.

For directed graph augmentation, Masuzawa, Hagihara, and Tokura in [MHT87] studied this problem when the input graph is a directed oriented tree. Their algorithm runs in $O(kn)$ time where k is the vertex-connectivity of the resulting graph. Jordán [Jor93a] gave a polynomial time approximation algorithm that uses no more than k extra edges for augmenting a $(k-1)$-vertex-connected directed graph to achieve k-vertex-connectivity. Very recently, Frank and Jordán [FJ93] gave a polynomial-time algorithm to solve the smallest vertex-connectivity augmentation problem on directed graphs exactly. Their algorithm increases the vertex-connectivity of a directed graph by any given

δ optimally. We do not know how to solve the smallest vertex-connectivity problem (both on undirected graphs and directed graphs) if we just want to satisfy the given connectivity requirement on a specified set of vertices.

2.5 Adding Edges to Meet Other Requirements

Most of the graph augmentation problems that we mentioned are for achieving various connectivity requirements. There are several results on achieving different requirements. Most of the output requirements are motivated by real-world applications.

Eswaran [Esw73] showed how to augment a directed graph such that the resulting graph is Eulerian by using a set of edges with the minimum total cost. Boesch, Suffel, and Tindell [BST77] studied this problem on undirected graphs when edges have a uniform cost. They specified the class of graphs that can be augmented to Eulerian graphs and an algorithm for finding such a smallest augmentation. Goodman and Hedetniemi [GH73] gave an $O(n)$ time algorithm to augment a tree such that the resulting graph is Hamiltonian.

Bokhari and Raza [BR84] studied the problem of adding at most one edge to each node. Under this paradigm, they gave polynomial algorithms for obtaining a graph with diameter less than or equal to $4\lceil\log\frac{n+2}{3}\rceil - 2$ from a tree, for obtaining a graph that contains a binary tree with diameter less than or equal to $2\lceil\log\frac{n+2}{3}\rceil$ from a chain, and for obtaining a graph that can emulate a shuffle-exchange network in constant time from a mesh.

Jain and Gopal [JG86] studied the problem of adding exactly one edge to the graph. Given a set of sources S and sinks T in the (undirected) graph, one endpoint of e must be in $S \cup T$ and the number of shortest paths from any vertex in S to any vertex in T is optimally increased by adding e.

Instead of developing an exact solution for this problem, they used a heuristic algorithm and no analysis was given.

Mo [Mo88] studied several smallest augmentation problems related to bounding the radius of a graph. Given a set of k vertices C and a value r, the problem of finding a smallest augmentation such that all vertices are within a radius of r from a vertex in C is called the *k-centrix radius r augmentation*. Mo proved that this problem is **NP**-hard for any $r \geq 2$ and gave an $O(kn)$ time algorithm for $r = 1$. Mo also solved the problem when the input graph is a tree. By extending the definition of reachability between two vertices, Mo gave two more algorithms for related problems. In addition to the above, Mo described the problem of adding a smallest set of edges such that the resulting graph contains k disjoint maximum matchings. This problem is **NP**-hard for "most" input graphs. Mo gave a polynomial time algorithm when the input graph is a forest.

For results on augmenting a graph such that the resulting graph is an interval graph, Kashiwabara and Fujisawa [KF79] first showed that it is **NP**-hard (even for the case when edges have uniform costs). Instead of finding an approximation solution, Ohtsuki, Mori, Kashiwabara and Fujisawa [OMKF81] developed an $O(nm)$ time algorithm for finding a minimal set of augmenting edges. No bound on the number of added edges was given.

2.6 Parallel Algorithms for Graph Augmentation

Several parallel algorithms have been developed for finding a smallest augmentation to connect a graph (that is, the problem of finding a spanning tree). Results can be found in Gibbons and Rytter [GR88], JáJá [JáJ92], Karp and Ramachandran [KR90], Quinn [Qui87], and the recent result of Chong and Lam [CL93].

Efficient parallel algorithms for finding smallest augmentations to achieve the following connectivity requirements can be found in Soroker [Sor88]. For finding a smallest augmentation to 2-edge-connect an undirected graph with n nodes and m edges, Soroker's parallel algorithm runs in $O(\log n)$ time on a CRCW PRAM using $O(\frac{(n+m)\alpha(m,n)}{\log n})$ processors, where $\alpha(m,n)$ is the inverse Ackermann function. This efficient parallel algorithm is a trivial parallelization of the linear time algorithm given in Eswaran and Tarjan [ET76]. For finding a smallest augmentation to strongly connect a directed graph with n nodes, his algorithm runs in $O(\log n)$ time on a CRCW PRAM using $O(M(n))$ processors, where $M(n)$ is the number of processors needed for a CRCW PRAM to compute the transitive closure of an n by n matrix in $O(\log n)$ time. (The realistic value for $M(n)$ is n^3, but there are various involved algorithms whose $M(n)$ is n^{2+c}, where c is a constant that is less than 1.) This efficient algorithm is a parallelization of the sequential algorithm described in Eswaran and Tarjan [ET76]. In this parallel algorithm Soroker gave an efficient parallel implementation for solving a special class of the matching problem. His parallel algorithm for finding a smallest set of directed edges whose addition makes a mixed graph strongly orientable runs in $O(\log n)$ on a CRCW PRAM using $O(M(n))$ processors.

Hsu and Ramachandran [HR91b] gave an $O(\log^2 n)$ time algorithm for finding a smallest biconnectivity augmentation on an EREW PRAM using a linear number of processors. Using similar techniques, we gave a parallel algorithm for finding a smallest triconnectivity augmentation with the same processor and time complexities. By using a technique to reduce from augmenting edge-connectivity to augmenting vertex-connectivity, Hsu and Ramachandran gave an $O(\log n)$ time CRCW algorithm for achieving 3-edge-connectivity using

$O(\frac{(n+m)\alpha(m,n)}{\log n})$ processors given the adjacency list of the input graph. These parallel algorithms will be described in Chapters 3, 4, 5, and 7 of this thesis. There is no efficient parallel algorithm known for finding a smallest augmentation to k-vertex-connect or k-edge-connect a graph, for any $k > 3$.

Chapter 3

Smallest Biconnectivity Augmentation

3.1 Introduction

In this chapter, we present an efficient parallel algorithm for finding a smallest augmentation to biconnect an undirected graph. In addition, we discover an error in the sequential algorithm of Rosenthal and Goldner [RG77]. We first give a corrected linear time sequential algorithm for the problem. Our efficient parallel algorithm is based on this corrected sequential algorithm. However we have to utilize several insights into the problem in order to derive the parallel algorithm. The algorithm runs in $O(\log^2 n)$ time using a linear number of processors on an EREW PRAM, where n is the number of vertices in the input graph. The work described in this chapter also appears in [HR91b].

3.2 Definitions

Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E. Let $\{E_i \mid 1 \leq i \leq k\}$ be a partition of E into a set of k disjoint subsets such that two edges e_1 and e_2 are in the same partition if and only if e_1 is equal to e_2 or there is a simple cycle in G containing e_1 and e_2 A vertex is *isolated* if it is not adjacent to any other vertex. Let q be the number of isolated vertices in G. Let $\{V_i \mid 1 \leq i \leq k+q\}$ be a collection of sets of vertices, where V_i is the set of vertices in E_i for each i, $1 \leq i \leq k$, and V_{i+k} contains only the ith isolated vertex for each i, $1 \leq i \leq q$. A vertex v is a *cutpoint* of a graph G if v appears in more than one vertex set V_i. G is *biconnected* if it has at

least 3 vertices and contains no cutpoint or isolated vertex. The subgraph $G_i = (V_i, E_i)$, $\forall i$, $1 \leq i \leq k$, is a *biconnected component* of G if V_i contains more than two vertices. Note that $E_i = \emptyset$, $k < i \leq k + q$, since V_i contains an isolated vertex. The subgraph $G_i = (V_i, E_i)$, $1 \leq i \leq k + q$, is called a *block* of G. Given an undirected graph G, we can define its *block graph* $blk(G)$ as follows. Each block and each cutpoint of G is represented by a vertex of $blk(G)$. The vertices of $blk(G)$ which represent blocks are called *b-vertices* and those representing cutpoints are called *c-vertices*. Two vertices u and v of $blk(G)$ are adjacent if and only if u is a *c*-vertex, v is a *b*-vertex, and the corresponding cutpoint of u is contained in the corresponding block of v or vice versa. It is well-known that $blk(G)$ is a forest and that if G is connected, $blk(G)$ is a tree. If $blk(G)$ is a tree, it is also called a *block tree*.

Let n_c be the number of *c*-vertices in $blk(G)$. A vertex v_i represents a *c*-vertex of $blk(G)$ and d_i is the degree of v_i. We assume that $d_i \geq d_{i+1}$, $1 \leq i < n_c$ throughout the discussion. For convenience, we define $a_i = d_i - 1$. If $blk(G)$ is a tree, let T be the rooted tree obtained from $blk(G)$ by rooting $blk(G)$ at the *b*-vertex which connects to v_1 and is on the path from v_1 to v_2. We use T_i to represent the subtree of T rooted at v_i for each i, $1 \leq i \leq n_c$, and we use T' to represent the subtree of T after deleting T_1. Let l_i be the number of leaves of T_i, $1 \leq i \leq n_c$. We also use T_v to represent the subtree rooted at a vertex v of $blk(G)$. The subgraph of T induced by deleting the vertex v is denoted by $T - v$.

In a forest, a vertex with degree 1 is a *leaf*. Let l be the number of leaves in $blk(G)$. For a graph G', we use l' to denote the number of degree-1 vertices in $blk(G')$. Let $d(v)$ be the degree of the vertex v in $blk(G)$ and let d be the largest degree of all **c-vertices** in $blk(G)$.

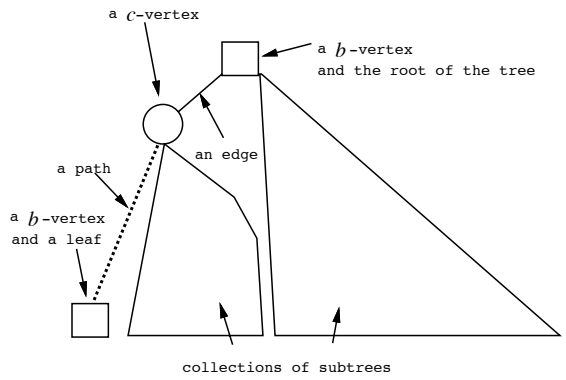


Figure 3.1: Notations for figures.

In figures, we use a rectangle to represent a b-vertex and a circle to represent a c-vertex. A line denotes an edge. A path in the block graph is represented by a thick dashed line while a polygon represents a collection of subtrees. These notations are shown in Figure 3.1.

We now give several definitions. Part of Definition 3.2.4 is from [RG77].

Definition 3.2.1 *A vertex v of $blk(G)$ is* **massive** *if and only if v is a c-vertex with $d(v) - 1 > \lceil \frac{l}{2} \rceil$. A vertex v of $blk(G)$ is* **critical** *if and only if v is a c-vertex with $d(v) - 1 = \lceil \frac{l}{2} \rceil$. The graph $blk(G)$ is critical if and only if there exists a critical c-vertex in $blk(G)$.*

Definition 3.2.2 *A block graph $blk(G)$ is* **balanced** *if and only if G is connected and does not contain a massive c-vertex. (Note that $blk(G)$ could have a critical c-vertex.) A graph G is balanced if and only if $blk(G)$ is balanced.*

Definition 3.2.3 (The leaf-connecting condition) *Two leaves u_1 and u_2 of $blk(G)$ satisfy the* **leaf-connecting condition** *if and only if u_1 and u_2 are in the same tree of $blk(G)$ and the path P from u_1 to u_2 in $blk(G)$ contains*

either (1) two vertices of degree more than 2, or
(2) one b-vertex of degree more than 3.

Definition 3.2.4 *Let v be a c-vertex of $blk(G)$. We call those components of $blk(G)-v$ that contain only one vertex of degree 1 in $blk(G)$* v-**chains** *[RG77]. A degree-1 vertex of $blk(G)$ in a v-chain is called a* v-**chain leaf.**

3.3 Main Lemmas

In this section, we present results that will be crucial in the development of our efficient parallel algorithm.

Lemma 3.3.1 *If $blk(G)$ has more than two c-vertices, then $a_1+a_2+a_3-1 \leq l$.*

Proof. Note that v_1 is a c-vertex with the largest degree. Vertex v_2 is a c-vertex with the largest degree among all c-vertices other than v_1. Vertex v_3 is a c-vertex with the largest degree among all c-vertices other than v_1 and v_2. Recall that if $blk(G)$ is a tree, we root $blk(G)$ at the b-vertex b which connects to v_1 and is on the path from v_1 to v_2. Let the rooted tree be T. Recall that T_i is the subtree of T rooted at v_i and l_i is the number of leaves in T_i. T' is the subtree obtained from T by removing T_1. Let l_x be the number of leaves in T'. *Case 1:* If v_3 is in T_1, then $l_1 \geq a_1 - 1 + a_3$ and $l_x \geq a_2$. This implies $l = l_1 + l_x \geq a_1 + a_2 + a_3 - 1$. *Case 2:* If v_3 is in T', but not in T_2, then $l_1 \geq a_1$ and $l_x \geq a_2 + a_3$. Thus $l = l_1 + l_x \geq a_1 + a_2 + a_3$. *Case 3:* If v_3 is in T_2, then $l_1 \geq a_1$ and $l_x \geq l_2 \geq a_2 - 1 + a_3$. This implies $l = l_1 + l_x \geq a_1 + a_2 + a_3 - 1$.

Suppose that $blk(G)$ is a forest and v_1, v_2, and v_3 are in different trees T_1, T_2, and T_3, respectively. If v_i is the only c-vertex in T_i, then $a_i = l_i - 1$.

Otherwise, $a_i \leq l_i$. Thus $a_1 + a_2 + a_3 \leq l_1 + l_2 + l_3 \leq l$. It is easy to prove the lemma for the case that $blk(G)$ is a forest and any two of v_1, v_2, and v_3 are in the same tree. □

Corollary 3.3.2 *If $blk(G)$ has more than two c-vertices, then $a_3 \leq \frac{l+1}{3}$.*

Proof. From the definition, we know that $a_1 \geq a_2 \geq a_3$. If $a_3 > \frac{l+1}{3}$, then $a_1 \geq a_2 \geq a_3 > \frac{l+1}{3}$ which implies $a_1 + a_2 + a_3 > \frac{l+1}{3} \cdot 3 = l + 1$. This is a contradiction to Lemma 3.3.1. □

Corollary 3.3.3 *There can be at most one massive vertex in $blk(G)$.*

Proof. The corollary is obviously true if there are less than two c-vertices in $blk(G)$. If $blk(G)$ has only two c-vertices v_1 and v_2, there is a b-vertex b^* in $blk(G)$ that connects to both v_1 and v_2. We root $blk(G)$ at b^*. Since there are only two c-vertices, the children of v_1 and v_2 are all leaves. We know that a_1 and a_2 are equal to the number of children of v_1 and v_2 respectively, thus $a_1 + a_2 = l$. Suppose v_1 is massive, then $a_1 > \frac{l}{2}$. Thus $a_2 < \frac{l}{2}$. If $blk(G)$ has more than two c-vertices and v_1 and v_2 are massive, then $a_1 + a_2 > l$. Since $a_3 \geq 1$, we have derived a contradiction to Lemma 3.3.1. □

Corollary 3.3.4 *If there is a massive vertex in $blk(G)$, then there is no critical vertex in $blk(G)$.*

Proof. The proof of Corollary 3.3.3 also applies here. □

Corollary 3.3.5 *There can be at most two critical vertices in $blk(G)$, if $l > 2$.*

Proof. The corollary is obviously true if $blk(G)$ has only one or two c-vertices. Assume that $blk(G)$ has more than two c-vertices. From Corollary refcoro:a3,

we know that $a_3 \leq \frac{l+1}{3}$. Since $\lceil \frac{l}{2} \rceil \geq \frac{l}{2} > \frac{l+1}{3}$, if $l > 2$, we know that v_3 cannot be critical if $l > 2$. □

Before introducing the next lemma, we have to study properties for updating the block tree. The following method for obtaining $blk(G')$ from $blk(G)$ is given in Rosenthal and Goldner [RG77].

Fact 3.3.6 *Given a graph G and its block tree $blk(G)$, adding an edge between two leaves u and v of $blk(G)$ creates a cycle C. Let G' be the graph obtained by adding an edge between u' and v' in G where u' and v' are non-cutpoint vertices in the blocks represented by u and v respectively. The following relations hold between $blk(G)$ and $blk(G')$. (1) Vertices and edges of $blk(G)$ that are not in the cycle C remain the same in $blk(G')$. (2) All b-vertices in $blk(G)$ that are in the cycle C contract to a single b-vertex b' in $blk(G')$. (3) Any c-vertex in C with degree equal to 2 is eliminated. (4) A c-vertex x in C with degree greater than 2 remains in $blk(G')$ with edges incident on vertices not in the cycle. The vertex x also attaches to the b-vertex b' in $blk(G')$.* □

An example of forming $blk(G')$ from $blk(G)$ is illustrated in Figure 3.2.

Lemma 3.3.7 *Let u_1 and u_2 be two leaves of $blk(G)$ satisfying the leaf-connecting condition (Definition 3.2.3). Let x_1 and x_2 be non-cutpoint vertices in blocks of G represented by u_1 and u_2 respectively. Let G' be the graph obtained from G by adding an edge between x_1 and x_2, and let P represent the path between u_1 and u_2 in $blk(G)$. The following three conditions hold. (1) $l' = l - 2$. (2) If v is a cutpoint in P with degree greater than 2 in $blk(G)$, then the degree of v decreases by 1 in $blk(G')$. (3) If v is a cutpoint in P with degree equal to 2, then v is eliminated in $blk(G')$.*

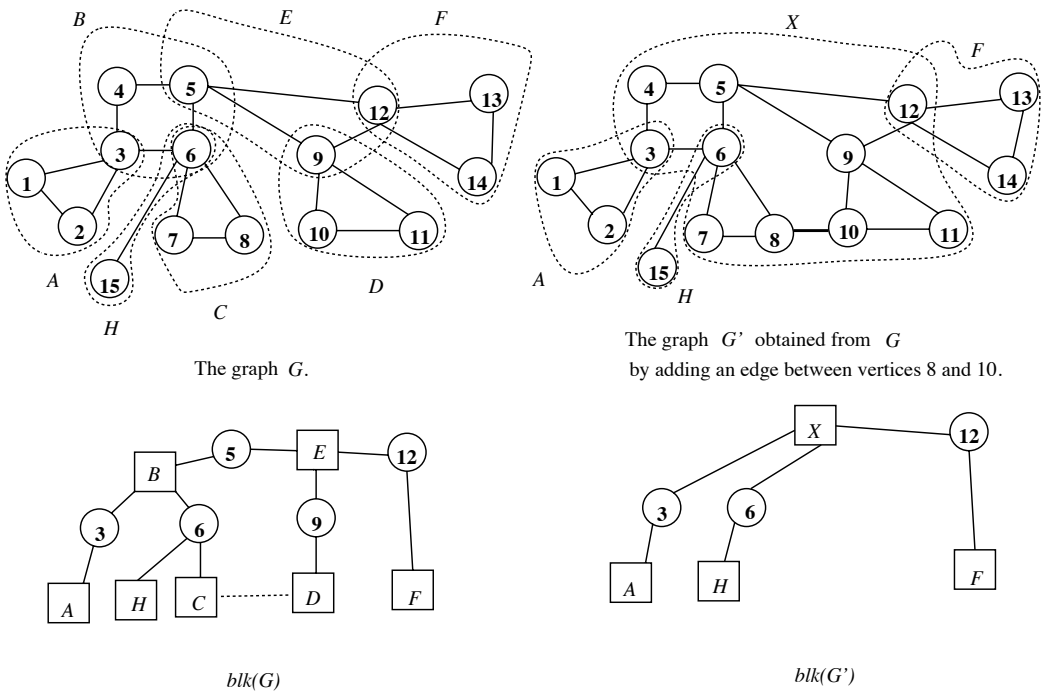


Figure 3.2: An example of obtaining $blk(G')$ from $blk(G)$. Vertices of G and G' circled with a dotted line are in the same block. For example, vertices 1, 2, and 3 of G are in block A. A vertex that appears in more than one block is a cutpoint. For example, vertex 3 appears in block A and B, thus it is a cutpoint. Vertices B, C, D, and E in $blk(G)$ are in a cycle if we add an edge between C and D. The cycle contracts into a new b-vertex X in $blk(G')$. The degree of a c-vertex in the cycle decreases by 1 in $blk(G')$, if the original degree is more than two. A degree-2 c-vertex in the cycle is eliminated in $blk(G')$.

Proof. Parts (2) and (3) of the lemma follow from parts (3) and (4) of Fact 3.3.6. We now prove part (1) of the lemma.

From part (2) in Fact 3.3.6, we know that every vertex of G that is in a component represented by a b-vertex in P is in a biconnected component Q of G'. Let Q be represented by a b-vertex b in $blk(G')$.

Case 1: Suppose that part (1) of the leaf-connecting condition (Definition 3.2.3) holds. Let w and y be two vertices of $blk(G)$ having degree more

than 2 in $blk(G)$ and let $blk(G')$ be rooted at b. In $blk(G)$, let w' be a vertex adjacent to w and y' be a vertex adjacent to y, with neither w' nor y' in P. The vertex b has at least two children, w' and y', in $blk(G')$ and hence cannot be a leaf. Since leaves u_1 and u_2 are eliminated in $blk(G')$ and no new leaf is created, $l' = l - 2$.

Case 2: Suppose that part (2) of the leaf-connecting condition (Definition 3.2.3) holds. Let w be a b-vertex of degree more than 3. We can find at least two c-vertices, y' and z', connected to w, but not in P. The same reasoning used in case 1 can now be applied. □

3.4 The Algorithm

The original linear time sequential algorithm in Rosenthal and Goldner [RG77] consists of three stages. However, we have discovered an error in stage 3 of the algorithm in [RG77]. We present a corrected version of that stage of the algorithm here. Our parallel algorithm follows the structure of the corrected sequential algorithm. The first two stages are easy to parallelize; we describe them in Sections 3.4.1 and 3.4.2. However, stage 3 is highly sequential. Most of our discussion concerns the corrected algorithm for stage 3 and its parallelization (Section 3.4.3).

We first state a lower bound on the number of edges needed to augment a graph to achieve biconnectivity.

Theorem 3.4.1 *Eswaran and Tarjan [ET76]*
Let G be an undirected graph with h connected components, and let q be the number of isolated vertices in $blk(G)$. Then at least $\max\{d + h - 2, \lceil \frac{l}{2} \rceil + q\}$ *edges are needed to biconnect G, if $q + l > 1$.* □

3.4.1 Stage 1

Theorem 3.4.2 *Rosenthal and Goldner [RG77]*
Let G be an undirected graph with h connected components. We can connect G by adding $h-1$ edges, which we may choose to be incident on non-cutpoint vertices in blocks corresponding to leaves or isolated vertices in $blk(G)$. □

Given $blk(G)$, it is easy to derive a parallel algorithm for stage 1 that runs optimally in $O(\log n)$ time on an EREW PRAM by using the Euler tour technique described in Tarjan and Vishkin [TV85]. The block graph can be updated by creating a new b-vertex b and two new c-vertices c_v and c_w for each new edge (v, w). We create edges from b to c_v and b to c_w. Let b_v and b_w be the two b-vertices in the block graph whose corresponding blocks contain v and w, respectively. We create edges from c_v to b_v and from c_w to b_w.

3.4.2 Stage 2

Theorem 3.4.3 *Rosenthal and Goldner [RG77]*
Let G be connected and let v^* be a massive vertex in G. Let $\delta = d - 1 - \lceil \frac{l}{2} \rceil$. We can find at least $2\delta + 2$ v^*-chains. Let Q be the set of v-chain leaves. By adding $2k, k \leq \delta$, edges to connect $2k+1$ vertices of Q, we can reduce both the degree of the massive vertex and the number of leaves in the block tree by k. □

Corollary 3.4.4 *Rosenthal and Goldner [RG77]*
Let G be connected and let v^* be a massive vertex in G. Let $\delta = d - 1 - \lceil \frac{l}{2} \rceil$ and let Q be the set of v^*-chain leaves. By adding 2δ edges to connect $2\delta + 1$ vertices of Q, we can obtain a balanced block tree. □

In stage 2, v^*-chain leaves can be found by first finding the number of leaves in each subtree rooted at a child of v^*. A leaf is in a v^*-chain if and only if

it is in a one-leaf subtree rooted at a child of v^*. Let Q be the set of vertices (excluding v^*) on cycles created by adding edges. The new block graph can be updated by merging vertices in Q into a single b-vertex b. Vertices b and v^* are connected by a new edge. These procedures can be optimally parallelized to run in time $O(\log n)$ on an EREW PRAM.

3.4.3 Stage 3

In this stage, we deal with a graph G where $blk(G)$ is balanced. The idea is to add an edge between two leaves y and z under the conditions that the path P between y and z passes through all critical vertices and the new block tree has two less leaves if $blk(G)$ has more than 3 leaves. Thus the degree of any critical vertex decreases by 1 and the tree remains balanced.

In Rosenthal and Goldner [RG77], $blk(G)$ is rooted at a b-vertex b^*. A path P is found that contains two leaves y and z such that if $blk(G)$ contains two critical vertices v and w, P contains both of them. If $blk(G)$ contains less than two critical vertices, P contains b^* and a c-vertex with degree d (recall that d is the maximum degree of any c-vertex). It is possible that in the case that $blk(G)$ is balanced with more than three leaves and less than two critical vertices, P contains only one vertex of degree more than 2. If we add an edge between the two end points of P, it is possible that the new block tree has only one less leaf. An example of this is shown in Figure 3.3. Thus the lower bound cannot be achieved by this method.

We now give a corrected version of stage 3 which runs in linear time. Our method is based on the proof of the tight bound given in Eswaran and Tarjan [ET76], but we add an additional step to handle the case $d = 2$ (that is, $a_1 = 1$); the analysis of this case is omitted in [ET76]. We present our revised version of stage 3 in Algorithm 3.1.

```
graph function seq_bca(graph G);
{* G has at least 3 vertices and blk(G) is balanced; l is the number of degree-1
vertices in blk(G); a_1 + 1 is the largest degree of all c-vertices in blk(G). *}
    tree T; vertex v, w, y, z, x_1, x_2;
    let T be blk(G) rooted at an arbitrary b-vertex;
    while l ≥ 2 do
      if a_1 = 1 then
        if l = 2 then let v be any c-vertex in T; w := v
        else {* l > 2 *}
    1.    let v be a b-vertex with degree > 2; {* Such a vertex
          must exist if l > 2 and a_1 = 1. *}
          w := v; {* This is the default value for w. *}
          if there exists a b-vertex other than v with degree > 2 then
    2.      find a degree > 2 b-vertex w, w ≠ v,
            such that w, v, and the root are in a path;
            {* such a path can be found by using an algorithm in [RG77] *}
          fi
        fi
      else {* a_1 > 1 *}
    3.  let v be a c-vertex with the largest degree in T;
        if ∃ a c-vertex other than v with degree > 2 then
    4.      find a degree > 2 c-vertex w, w ≠ v,
            such that w, v, and the root are in a path;
            {* such a path can be found by using an algorithm in [RG77] *}
        else {* v is the only c-vertex with degree > 2 *}
          w := v; {* This is the default value for w. *}
          if ∃ a b-vertex in T with degree greater than 2 then
    5.      find a degree > 2 b-vertex w, w ≠ v,
            such that w, v, and the root are in a path;
            {* such a path can be found by using an algorithm in [RG77] *}
          fi
        fi
      fi;
    6. find two leaves y and z such that y, v, w, and z are in a path;
      find a non-cutpoint vertex x_1 in the corresponding block of G represented by y;
      find a non-cutpoint vertex x_2 in the corresponding block of G represented by z;
      add an edge between x_1 and x_2; update the block graph T
    od;
    return G
end seq_bca;
```

Algorithm 3.1: Corrected sequential algorithm for finding a smallest biconnectivity augmentation.

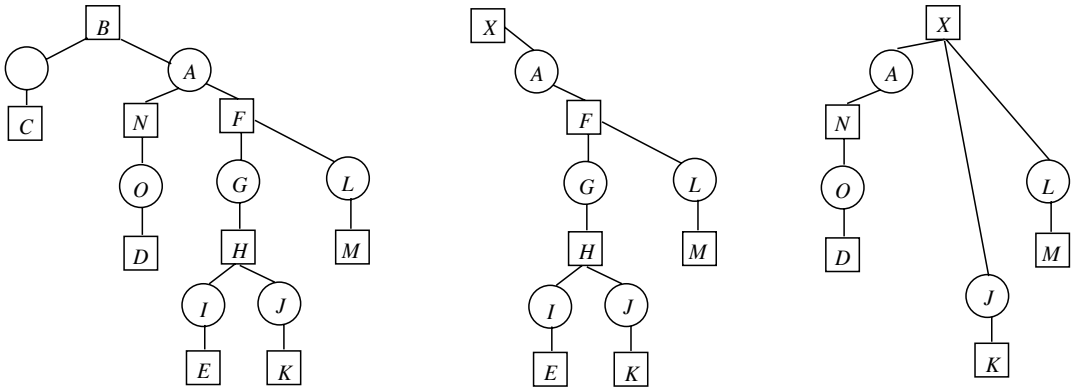


Figure 3.3: A counter example for the linear time sequential algorithm given by Rosenthal and Goldner. The left tree is $blk(G)$ rooted at B. Vertex A is the c-vertex with the largest degree. The middle tree is the new block tree after connecting two non-cutpoint vertices of G in the corresponding blocks represented by C and D. The number of leaves decreases by 1. The right tree is the new block tree after connecting two non-cutpoint vertices of G in the corresponding blocks represented by C and E. The number of leaves decreases by 2. The pair C and D could be chosen by the algorithm given by Rosenthal and Goldner while the pair C and E can be chosen to reduce the number of leaves by two.

In steps 2, 3, and 4 of algorithm `seq_bca`, we use a procedure described in [RG77] to find a vertex w with degree > 2 such that w, the root, and a given vertex v are in a path of the rooted block tree. The procedure is as follows. We first traverse the (simple) path P_1 from v towards the root. The first vertex we meet with degree > 2 is w. If there is no such vertex in P_1 and the degree of the root is > 1, let q be a child of the root such that q is not in P. We traverse one (simple) path P_2 from q to a leaf without passing the root. The first vertex we meet with degree greater than 2 is w. If we cannot a vertex with degree > 2 in P_1 and P_2, then we trace any (simple) path P_3 starting from v towards a leaf without passing any ancestor of v. The first vertex we meet with degree greater than 2 is w. It is well-known that if there exists a vertex other than v with degree greater than 2 in a tree, then this procedure will find one.

Claim 3.4.5 *If $blk(G)$ is balanced, we can biconnect G by adding $\lceil \frac{l}{2} \rceil$ edges using algorithm* `seq_bca`.

Proof. We first discuss the case where $blk(G)$ has more than 3 leaves. In this case, a critical vertex must have degree more than 2.

Case 1: If $blk(G)$ has two critical vertices v and w, it is known that the rest of the vertices have degree less than or equal to 2 [RG77]. Algorithm `seq_bca` finds the first critical vertex in step 3 and the second critical vertex in step 4.

Case 2: If $blk(G)$ has only one critical vertex v, algorithm `seq_bca` finds it in step 3. Because $blk(G)$ is balanced and $l > 3$, there must exist another vertex w with degree more than 2. Otherwise v is massive. Algorithm `seq_bca` finds w in step 4 or step 5.

Case 3: The block tree $blk(G)$ has no critical vertex. Then either there is only one vertex (which must be a b-vertex) with degree more than 3 or there are two vertices with degree more than 2. If there is only one vertex v with degree more than 3, algorithm `seq_bca` finds v in step 1. Suppose there are two vertices v and w with degree more than 2 in the block tree. If v and w are both c-vertices, algorithm `seq_bca` finds them in steps 3 and 4, respectively. If v and w are both b-vertices, algorithm `seq_bca` finds them in steps 1 and 2, respectively. If one of v and w is a c-vertex and the other one is a b-vertex, algorithm `seq_bca` finds the c-vertex in step 3 and the b-vertex in step 5.

In all three cases, we can find two vertices of degree more than 2 or a b-vertex of degree more than 3. Thus by Lemma 3.3.7, the number of leaves in the new block tree reduces by 2. Because v and w are the possible critical vertices, we reduce the value of d by 1. Thus the block tree remains balanced. Hence, the algorithm can achieve the lower bound in Eswaran and Tarjan [ET76].

For the case $l = 3$, we can reduce $blk(G)$ to a new block tree with two leaves by picking any pair of leaves in $blk(G)$ and connecting them. A block tree of 2 leaves can be reduced to a single vertex by connecting the two leaves. Thus the claim is true. □

Claim 3.4.6 *Algorithm* `seq_bca` *runs in* $O(n + m)$ *time.*

Proof. Using the same data structure as Rosenthal and Goldner [RG77] and similar techniques, we can implement our algorithm in linear time. □

In the rest of this section, we describe an efficient parallel algorithm for stage 3. Recall that the sequential algorithm adds one edge at a time and keeps adding edges until the block tree becomes a single vertex. In our parallel algorithm, however, we will find several pairs of leaves such that the path between any such pair of leaves passes through all critical c-vertices. Thus the degrees of critical vertices in the new block tree decrease by the number of edges added to the original block tree. These pairs also satisfy the leaf-connecting condition (Definition 3.2.3), which guarantees that the number of leaves in the new block tree decreases by twice the number of edges added. The following Lemma 3.4.7 tells us that the addition of several edges in parallel as outlined above is a valid strategy.

Lemma 3.4.7 *Let G be a graph whose block graph is balanced and let G' be the graph obtained from G by adding a set of k edges $\mathcal{A} = \{(s_1, t_1), \ldots, (s_k, t_k)\}$. For each i, $0 \leq i \leq k$, let G_i be the graph obtained from G by adding the set of edges $\{(s_1, t_1), \ldots, (s_i, t_i)\}$. Let s'_i and t'_i, $0 \leq i \leq k$, be the b-vertices in $blk(G_i)$ whose corresponding blocks contain s_i and t_i, respectively. If the following two conditions hold (1) s'_{i+1} and t'_{i+1} satisfy the leaf-connecting condition in G_i, $1 \leq i < k$ and (2) the path between s'_{i+1} and t'_{i+1} in $blk(G_i)$ passes through all*

critical vertices in $blk(G_i)$, then (1) $blk(G')$ remains balanced and (2) the value of the lower bound given in Theorem 3.4.1 applied to G' is k less than the same lower bound applied to G.

Proof. We obtain the same block graph for G_k no matter in what sequence we choose to add these k edges, since there is a unique block graph for each graph G. Thus $blk(G') = blk(G_k)$. Since $blk(G_i)$ is balanced for $1 \leq i \leq k$, $blk(G')$ is balanced. We know that the value of the lower bound given in Theorem 3.4.1 applied to G_i is 1 less than the value of the same lower bound applied to G_{i-1}, $1 \leq i \leq k$, where $G_0 = G$. Hence the value of the lower bound given in Theorem 3.4.1 applied to G' is k less than the value of the same lower bound applied to G. □

From Theorem 3.4.1 and Claim 3.4.5, we know that exactly $\lceil \frac{l}{2} \rceil$ edges must be added to biconnect G if $blk(G)$ is balanced. That is, we have to eliminate l leaves during the computation. Our parallel algorithm runs in stages with at least $\frac{1}{4}$ of the current leaves eliminated in parallel time $O(\log n)$ using a linear number of processors during each stage. We call this subroutine $O(\log n)$ times to complete the augmentation.

Recall that $a_i + 1$ is equal to the degree of the ith c-vertex v_i and $a_i \geq a_{i+1}$. The subtree T' is obtained from T by deleting the subtree rooted at v_1. Let $U_i = \{u \mid u \text{ is the leftmost leaf of } T_y, \text{ where } y \text{ is a child of } v_i\}$. For example, the leaves in U_1 are illustrated as shadowed rectangles in Figure 3.4.

Depending on the degree distribution of vertices in the block tree, the parallel algorithm for stage 3 is divided into 2 cases. In case 1, $a_1 > \frac{l}{4}$. We have a c-vertex with a high degree. We pick the first $\min\{a_1 - 1, \lceil \frac{l}{2} \rceil - a_3\}$ leaves in U_1 and call them W_1. Leaves in W_1 are matched with the first $\min\{|W_1|, |U_2| - 1\}$ leaves in U_2. Unmatched leaves in W_1, if any, are

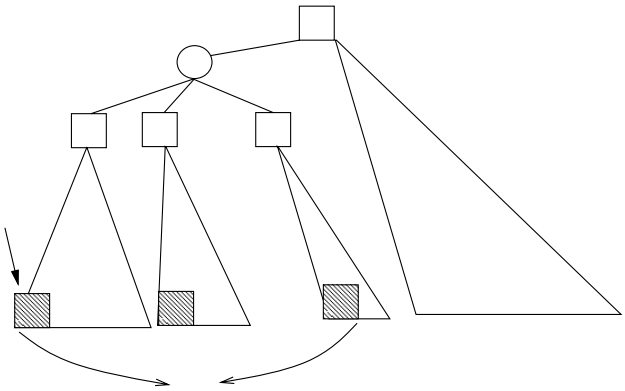


Figure 3.4: Each shadowed rectangle represents the leftmost leaf in a subtree rooted at a child of v_1. Leaves in U_1 consist of leftmost leaves in every subtree rooted at a child of v_1.

matched with all remaining leaves but one in T' before being properly matched within themselves, if necessary. In case 2, $a_1 \leq \frac{l}{4}$. There is no c-vertex with a large degree. We show that we can find a vertex u^* with approximately the same number of leaves in each subtree rooted at a child of u^*. If u^* is a b-vertex, a suitable number of leaves between subtrees rooted at children of u^* are matched. Otherwise, u^* is a c-vertex and a suitable number of subtrees rooted at children of u^* are first merged into a single subtree rooted at u^*. Then leaves in the merged subtree are matched with leaves outside.

The algorithm first finds the matched pairs of leaves in each case. Then we add edges between matched pairs of leaves and update the block tree at the end of each case. The block tree and the sequence of cutpoints $v_1, \ldots, v_{n_c}$ will not be changed during the execution of each case.

We now describe the two cases in detail.

Case 1: $a_1 > \frac{l}{4}$ We root the block tree at the b-vertex b^* which is adjacent to v_1 and is on the path from v_1 to v_2. Let v_1 be the leftmost child of b^*. We permute the children of v_1 in non-increasing order (from left to right) of

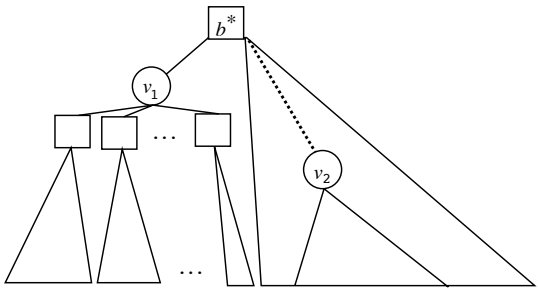


Figure 3.5: A normalized tree. Vertex v_1 is a c-vertex with the largest degree. Vertex v_2 is a c-vertex with a degree larger than or equal to any other c-vertices in $T - v_1$. We permute the children of v_1 in non-increasing order (from left to right) of the number of leaves in subtrees rooted at them.

the number of leaves in subtrees rooted at them. We will call this procedure *tree-normalization* and the resulting tree T. Figure 3.5 illustrates a normalized tree.

Recall that U_1 is the set of leftmost leaves in subtrees rooted at children of v_1. We select the first (from left to right) $\min\{a_1 - 1, \lceil \frac{l}{2} \rceil - a_3\}$ leaves from U_1 and call the set W_1. The order of the leaves as specified in the original tree is preserved. There are four phases for this case. In phases 1 and 2, leaves in W_1 are matched with leaves not in T_1. In phase 3, leaves in W_1 are matched with leaves in T_1 excluding those in W_1. In phase 4, the remaining leaves in W_1 are matched with themselves. The algorithm executes phases 1 through 4 in turns and keeps on iterating until there is no leaf in W_1 left to be matched.

We now describe the four phases in detail. After the description, we give the overall parallel algorithm for case 1 and prove that it eliminates a constant fraction of the leaves while maintaining the lower bound described in Theorem 3.4.1.

Phase 1: All leaves but the rightmost one in U_2 are matched with the rightmost $a_2 - 1$ leaves of W_1. The matched leaves are removed from W_1. An

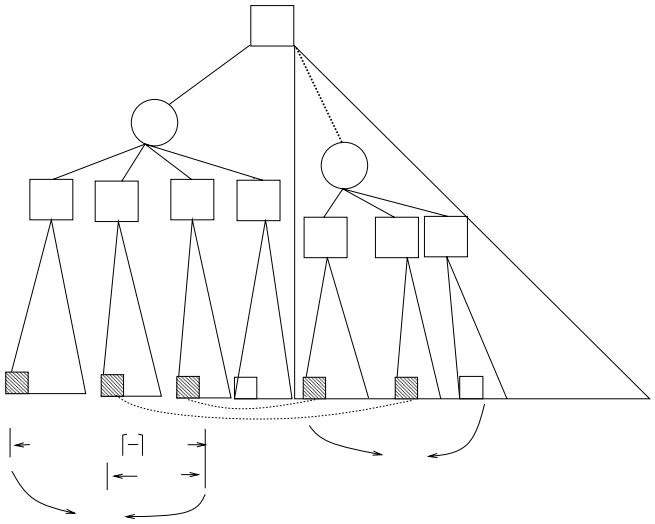


Figure 3.6: Pairs of matched leaves found in phase 1 of case 1 are connected by dotted lines. The set W_1 consists of the leftmost leaves from the first $\min\{a_1-1, \lceil \frac{l}{2} \rceil - a_3\}$ subtrees rooted at children of v_1. All of the leftmost leaves in subtrees rooted at children of v_2 are in the set U_2, and all except the rightmost leaf in U_2 are matched (if possible).

```
set of pairs of vertices function phase1(modifies set of vertices W1, U2);
    set of pairs of vertices L; vertex u, v;
    L := ∅; {* L is the set of matched pairs. *}
    number leaves in W1 from right to left starting from 1;
    number leaves in U2 from left to right starting from 1;
    k := min{|U2| − 1, |W1|};
    pfor i = 1 .. k do
        u := the ith leaf in W1; remove u from W1;
        v := the ith leaf in U2; remove v from U2;
        L := L ∪ {(u, v)}
    rofp;
    return L
end phase1;
```

Algorithm 3.2: Parallel algorithm to handle phase 1 of case 1.

example of the pairs of leaves matched in phase 1 is given in Figure 3.6. See Algorithm 3.2.

Phase 2: We match all remaining leaves but one in T' with the rightmost leaves of W_1 and remove matched leaves from W_1. See Algorithm 3.3. An

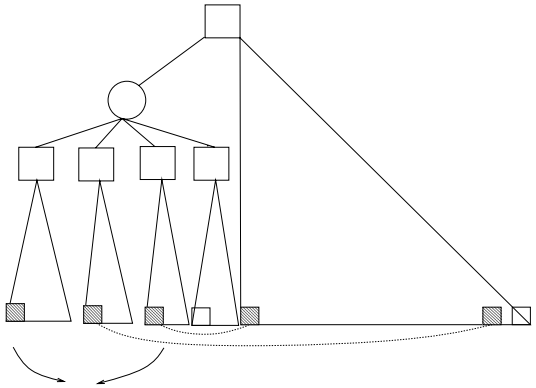


Figure 3.7: Pairs of matched leaves found in phase 2 of case 1 are connected by dotted lines. Recall that T_1 is the subtree of T rooted at v_1. The subtree T' is obtained from T by deleting T_1. The set W_1 consists of the leftmost leaves in the first $\min\{a_1 - 1, \lceil \frac{l}{2} \rceil - a_3\} - a_2 + 1$ subtrees rooted at children of v_1. Leaves in W_1 are matched with all but the rightmost leaf in T'.

```
set of pairs of vertices function phase2(modifies set of vertices W1, tree T');
    set of pairs of vertices L; vertex u, v;
    L := ∅; {* L is the set of matched pairs. *}
    number leaves in W1 from right to left starting from 1;
    number leaves in T' from left to right starting from 1;
    k := min{the number of leaves in T' minus 1, |W1|};
    pfor i = 1 .. k do
        u := the ith leaf in W1; remove u from W1; v := the ith leaf in T';
        L := L ∪ {(u, v)}
    rofp;
    return L
end phase2;
```

Algorithm 3.3: Parallel algorithm for phase 2 of case 1.

example of the pairs of leaves matched in phase 2 is given in Figure 3.7.

Phase 3: Recall that T is the original block tree before phase 1, l is the number of leaves in T, v_1 is a c-vertex with the largest degree in T, T_1 is the subtree of T rooted at v_1, l_1 is the number of leaves in T_1, T' is the tree obtained from T by removing T_1, and $U_1 = \{u \mid u$ is the leftmost leaf of T_y, where y is a child of $v_1\}$. Note that there are $\min\{a_1 - 1, \lceil \frac{l}{2} \rceil - a_3\} - (l - l_1 - 1)$ leaves remaining in W_1. Leaves in W_1 come from the first $|W_1|$ members (from

left to right) of U_1. Let the set of v_1-chain leaves in W_1 be Q_1. We denote by Q_2 the set of leaves other than the rightmost one of each subtree rooted at a child of v_1. (Note that $Q_1 \cap Q_2 = \emptyset$.) In this phase, we match all leaves in Q_1 (i.e., all v_1-chain leaves in W_1) with an equal number of leaves in Q_2. Leaves in W_1 that are matched in phase 3 (Q_1 and $W_1 \cap Q_2$) are removed from W_1. See Algorithm 3.4.

Claim 3.4.8 shows that we can always find enough leaves in Q_2 to match all leaves in Q_1.

Claim 3.4.8 *$|Q_2| \geq |Q_1|$, if $l > 3$.*

Proof. If $|Q_1| = 0$, the claim is true. Let $|Q_1| > 0$. Recall that there is only one unmatched leaf s left in T' after phase 2. Let T^* be the block tree obtained from T by adding edges between matched pairs of leaves found in phase 1 and phase 2. We root T^* at the b-vertex b^* which is adjacent to v_1 and is on the path from v_1 to s. Let r be the number of subtrees rooted at a child of v_1 in T^* with more than one leaf. Let y be the number of v_1-chain leaves not in Q_1. The notations used in this proof are shown in Figure 3.8.

The total number of leaves in T^* is equal to $|Q_1| + |Q_2| + r + y + 1$ if $|Q_1| > 0$. The degree of v_1 in T^* is equal to $|Q_1| + r + y + 1$. Since T^* is balanced (for a proof, see Claim 3.4.10 at the end of this section), v_1 is not massive and hence

$$|Q_1| + r + y \leq \left\lceil \frac{|Q_1| + |Q_2| + r + y + 1}{2} \right\rceil.$$

Thus

$$\begin{aligned} & 2|Q_1| + 2r + 2y \leq |Q_1| + |Q_2| + r + y + 2 \\ \Rightarrow \quad & |Q_1| + r + y - 2 \leq |Q_2|. \end{aligned}$$

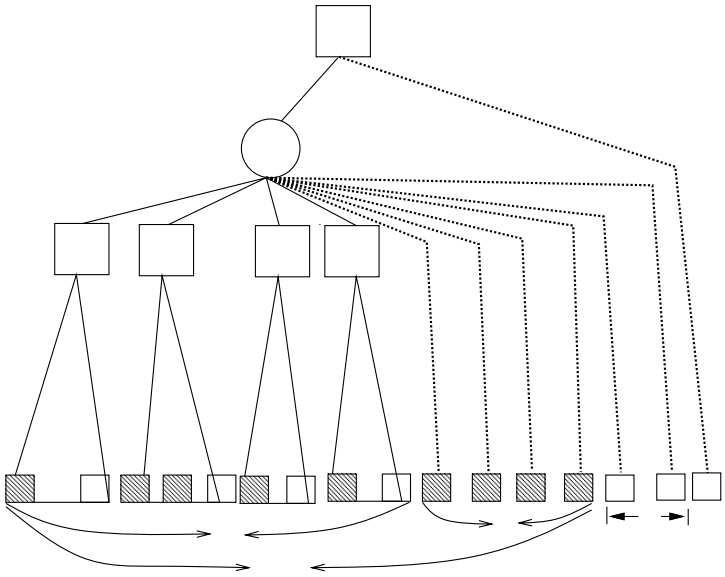


Figure 3.8: Notations used in the proof of a claim used in phase 3 of case 1. The tree shown is T^*, the updated block tree obtained by adding edges between pairs of matched leaves found in phase 1 and phase 2. The set Q_2 consists of all but the rightmost leaf in each subtree rooted at a child of v_1. The set Q_1 consists of v_1-chain leaves in W_1 after phase 2. The number of subtrees rooted at a child of v_1 with more than one leaf is r. The number of v_1-chain leaves not in Q_1 is y.

We know that $r \geq 1$, otherwise v_1 is massive if $l > 3$. It is also true that $y \geq 1$ if $|Q_1| > 0$. Thus $|Q_1| \leq |Q_2|$. □

Phase 4: The remaining leaves of W_1 that are not matched during phase 3 are matched within themselves. If the number of remaining leaves in W_1 is odd, we match one of them with the rightmost leaf in the subtree rooted at v_1. See Algorithm 3.5. An example of the pairs of leaves matched in phase 4 is given in Figure 3.9.

See Algorithm 3.6 for the complete algorithm for case 1.

Claim 3.4.9 *The number of matched pairs k in case 1 satisfies $\lceil \frac{l}{2} \rceil - a_3 \geq k \geq \frac{l}{8}$, if $l > 3$.*

```
set of pairs of vertices function phase3(modifies set of vertices Q1, Q2);
    set of pairs of vertices L; vertex u, v;
    L := ∅; {* L is the set of matched pairs. *}
    number leaves in Q2 from right to left starting from 1;
    number leaves in Q1 from 1 to |Q1| in arbitrary order;
    k := |Q1|;
    pfor i = 1 .. k do
        u := the ith leaf in Q2; remove u from Q2;
        v := the ith leaf in Q1; remove v from Q1;
        L := L ∪ {(u, v)}
    rofp;
    return L
end phase3;
```

Algorithm 3.4: Parallel algorithm for phase 3 of case 1.

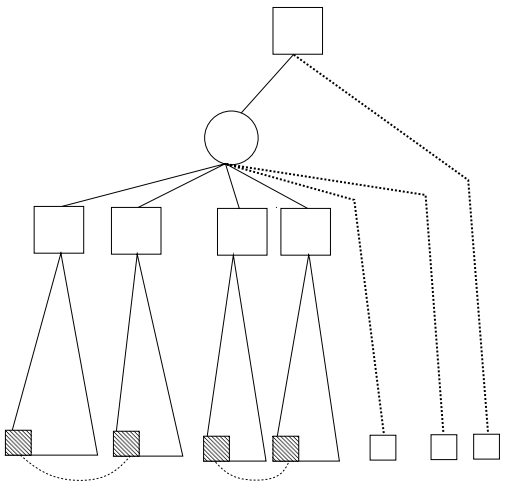


Figure 3.9: Illustrating phase 4 of case 1. The remaining leaves in W_1 are matched with themselves.

```
set of pairs of vertices function phase4(modifies set of vertices W1, tree T);
    set of pairs of vertices L; vertex u, v;
    L := ∅; {* L is the set of matched pairs. *}
    number leaves in W1 in arbitrary order from 1 to |W1|;
    k := ⌈|W1|/2⌉;
    pfor i = 1 .. k do
        u := the (2·i − 1)th leaf in W1; remove u from W1;
        if 2·i ≤ |W1| then v := the (2·i)th leaf in W1; remove v from W1
        else {* 2·i > |W1| *} v := the rightmost leaf in the subtree rooted at v1
        fi;
        L := L ∪ {(u, v)}
    rofp;
    return L
end phase4;
```

Algorithm 3.5: Parallel algorithm for phase 4 of case 1.

```
set of pairs of vertices function case1(tree T);
    set of pairs of vertices L; set of vertices W1, Q1, Q2; vertex b*; tree T';
    root T at the b-vertex b* which is adjacent to v1 and is on the path from v1 to v2;
    let v1 be the leftmost child of b* in T;
    permute the children of v1 in non-increasing order (from left to right)
    of the number of leaves in subtrees rooted at them;
    W1 := the first (from left to right) min{a1 − 1, ⌈l/2⌉ − a3} leaves of U1;
    L := phase1(W1, U2); {* L is the set of matched pairs. *}
    if W1 ≠ ∅ then L := L ∪ phase2(W1, T') fi;
    T1' := the subtree of T1 with the first |W1| subtrees rooted at children of v1;
    Q1 := the set of v1-chain leaves in Tv1;
    Q2 := {u | u is a non-leftmost leaf of Ty, where Ty has more than 1 leaf
    and y is a child of v1 in T1'};
    if W1 ≠ ∅ then L := L ∪ phase3(Q1, Q2) fi; W1 := W1 ∩ Q2;
    if W1 ≠ ∅ then L := L ∪ phase4(W1, T) fi;
    return L
end case1;
```

Algorithm 3.6: Parallel algorithm for case 1.

Proof. Let $z = \min\{a_1 - 1, \lceil \frac{l}{2} \rceil - a_3\}$. If the procedure does not execute phases 3 and 4, we match z pairs. Because $a_1 - 1 \geq \frac{l}{4}$ and $\lfloor \lceil \frac{l}{2} \rceil - a_3 \rfloor \geq \lfloor \frac{l}{8} \rfloor$ for $l > 3$ (Corollary 3.3.2), we know that $z \geq \frac{l}{8}$, if $l > 3$. Otherwise, in the worst case, we match only $a_2 - 1$ pairs during phase 1 and phase 2. A pair of vertices matched during phase 3 or 4 might be both members of W_1. Thus

$$k \geq a_2 - 1 + \left\lceil \frac{z - a_2 + 1}{2} \right\rceil \geq \frac{z + a_2 + 1}{2}.$$

If $z = \lceil \frac{l}{2} \rceil - a_3$, then $k \geq \frac{\lceil \frac{l}{2} \rceil - 1}{2}$, which is greater than or equal to $\frac{l}{8}$ if $l > 3$ and k is an integer. If $z = a_1 - 1$, then $k \geq \lceil \frac{a_1 - 1}{2} \rceil$. Because a_1 is greater than $\frac{l}{4}$, k is greater than or equal to $\frac{l}{8}$. □

Claim 3.4.10

(1) Each pair of matched vertices found in function `case1` *satisfies the leaf-connecting condition (Definition 3.2.3).*

(2) Let us place an edge between each matched pair found in function `case1` *sequentially, and update the block graph each time we add an edge. Critical vertices, if any, of the block graph are on the path between the endpoints of each edge placed.*

Proof. From part (4) in Fact 3.3.6, the degrees of v_1 and v_2 decrease only by 1 by adding an edge between a pair of matched vertices. Let us consider paths in the block tree between pairs of vertices matched in each phase. We show that we can find at least two vertices with degree more than 2 in each path.

Phase 1: The path between each pair of matched vertices passes through v_1 and v_2. The degrees of v_1 and v_2 are at least 3.

Phase 2: The path between each pair of matched vertices passes through v_1 and the root b^*. The degrees of v_1 and b^* are at least 3.

Phase 3 and phase 4: The path P between each pair of matched vertices passes through v_1. The degree of v_1 is at least 3. The path P also passes through a child u of v_1 where the subtree rooted at u has more than one leaf. Thus the degree of u is more than 2.
Thus the leaf-connecting condition (Definition 3.2.3) holds for each pair of matched vertices.

Because we only add $\min\{a_1-1, \lceil\frac{l}{2}\rceil-a_3\}$ edges during case 1, v_3 and thus v_i, such that $i \geq 4$, does not become critical. From the previous discussion, the path between each pair of matched vertices passes through v_1 and v_2, the only two possible critical vertices, during phase 1. We reduce degrees of possible critical vertices by one by adding one new edge between each pair of matched vertices. If we match any pair of vertices after phase 1, the degree of v_2 is at most 2 and the degree of v_1 is at least 3. Thus v_1 is the only possible critical vertex. The path between each pair of matched vertices passes through v_1 after phase 1. We reduce the degree of the possible critical vertex, v_1, by one by adding one new edge between any pair of matched vertices. Thus the claim holds. □

Corollary 3.4.11 *Let k be the number of matched pairs found in function* `case1`. *Let G' be the resulting graph obtained from the current graph G by adding a new edge between each matched pair of leaves. The value of the lower bound given in Theorem 3.4.1 applied to G' is k less than the value of the same lower bound applied to G, and $blk(G')$ remains balanced. Let l be the number of leaves in $blk(G)$. The number of leaves in $blk(G')$ is at most $\frac{3l}{4}$, if $l > 3$.*

Proof. From part (1) in Claim 3.4.10, the number of leaves in $blk(G')$ is $2k$ less than the number of leaves in $blk(G)$. Since $k \geq \frac{l}{8}$ if $l > 3$ (Claim 3.4.9), the

number of leaves in $blk(G')$ is at most $\frac{3l}{4}$ if $l > 3$. From part (2) in Claim 3.4.10, the block graph of each intermediate graph remains balanced even if we place a new edge between each matched pair of leaves found in function `case1` one by one. By Lemma 3.4.7, we know that the value of the lower bound given in Theorem 3.4.1 applied to G' is k less than the value of the same lower bound applied to G and $blk(G')$ remains balanced. □

Case 2: $a_1 \leq \frac{l}{4}$ In this case, we take advantage of the fact that no c-vertex has a large degree. Because there is no critical c-vertex, the algorithm can add at least $\lceil \frac{l}{2} \rceil - a_1$ edges between leaves that satisfy the leaf-connecting condition (Definition 3.2.3) without worrying about whether the path between them passes through a critical c-vertex. This gives a certain degree of freedom for us to choose the matched pairs. We first root the block tree such that no subtree, other than the ones that are rooted at the root, has more than half of the total number of leaves.

Given any rooted tree T, we use l_v to denote the number of leaves in the subtree rooted at a vertex v. The following lemma shows that we can reroot T at a vertex u^* such that no subtree rooted at a child of u^* has more than half of the total number of leaves.

Lemma 3.4.12 *Given a rooted tree T, there exists a vertex u^* in T such that $l_{u^*} > \frac{l}{2}$, and none of the subtrees rooted at children of u^* has more than $\frac{l}{2}$ leaves.*

Proof. We permute the children of each non-leaf vertex v from left to right in non-increasing order of the number of leaves in the subtrees rooted at them. Let us consider the leftmost path P of the tree T. It is obvious that there exists such a vertex u^* in P. □

We root the block tree at u^* and permute the children of u^* from left to right in non-increasing order of the number of leaves in subtrees rooted at them. Let the rooted tree be T. Let u_i, $1 \leq i \leq r$, be the children (from left to right) of u^*, and x_i be the number of leaves in the subtree rooted at u_i. Note that $x_i \leq \frac{l}{2}$, for each i. There are two subcases depending on whether u^* is a b-vertex or a c-vertex. We describe the two subcases in detail in the following paragraphs.

Subcase 2.1: u^* is a b-vertex We show that we can "evenly" partition subtrees rooted at children of the root into two sets such that we can match leaves between the two partitions. See Algorithm 3.7. We first give a claim to show how to perform the partition.

Claim 3.4.13 *There exists p such that $1 \leq p < r$ and $\frac{l}{2} \geq \sum_{i=1}^{p} x_i > \frac{l}{4}$.*

Proof. We know that $x_i \geq x_{i+1}$, $1 \leq i < r$, and $x_i \leq \frac{l}{2}$, $1 \leq i \leq r$. Thus there exists p, $1 \leq p < r$, such that

$$\sum_{i=1}^{p} x_i \leq \frac{l}{2} \text{ and } \sum_{i=1}^{p+1} x_i > \frac{l}{2}.$$

Because $x_i \geq x_{i+1}$, $1 \leq i < r$, we know that $\sum_{i=1}^{p} x_i > \frac{1}{2}(\frac{l}{2})$. □

The notations used for this subcase are illustrated in Figure 3.10.

Corollary 3.4.14 $\sum_{i=1}^{p} x_i \leq l - \sum_{i=1}^{p} x_i$. □

We match $\min\{(\sum_{i=1}^{p} x_i) - 1, \lceil \frac{l}{2} \rceil - a_1\}$ leaves in subtrees T_{u_i}, $1 \leq i \leq p$, with leaves outside them. From Claim 3.4.13 and Corollary 3.4.14, we know that the matching can be done.

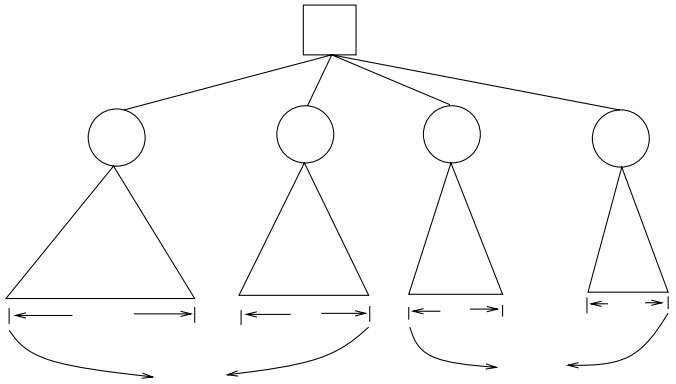


Figure 3.10: The notations used in case 2.1. The number of leaves in the subtree rooted at u_i is x_i. We find the largest p such that the total number of leaves in the first p subtrees rooted at children of the root is greater than $\frac{l}{4}$, but less than or equal to $\frac{l}{2}$. Leaves in the first p subtrees rooted at children of b^* are in Z_1. The set Z_2 consists of the rest of the leaves in the tree.

```
set of pairs of vertices function case2_1(tree T);
{* l is the number of leaves in T. *}
    integer p; set of pairs of vertices L; set of vertices Z1, Z2; vertex u, v;
    let ui be the ith (from left to right) child of the root;
    let xi be the number of leaves in the subtree rooted at ui;
    find the largest integer p such that sum_{i=1}^{p} xi <= l/2, but sum_{i=1}^{p+1} xi > l/2;
    L := ∅; {* L is the set of matched pairs. *}
    Z1 := the set of leaves in the subtrees rooted at ui, 1 <= i <= p;
    Z2 := the set of leaves in the subtrees rooted at ui, i > p;
    number leaves in Z1 in arbitrary order from 1 to |Z1|;
    number leaves in Z2 in arbitrary order from 1 to |Z2|;
    k := min{(sum_{i=1}^{p} xi) - 1, ceil(l/2) - a1};
    pfor i = 1 .. k do
        u, v := the ith vertex in Z1 and Z2, respectively;
        L := L ∪ {(u, v)}
    rofp;
    return L
end case2_1;
```

Algorithm 3.7: Parallel algorithm for case 2.1.

Corollary 3.4.15 *The number of matched pairs k in case 2.1 satisfies $\lceil \frac{l}{2} \rceil - a_1 \geq k > \frac{l}{4}$, if $l > 3$.* □

Claim 3.4.16 *Any matched pair found in function* `case2_1` *satisfies the leaf-connecting condition (Definition 3.2.3), if $l > 3$.*

Proof. Consider the path P between a pair of matched leaves u and v. Let u be a leaf in a subtree rooted at vertex u_x, $1 \leq x \leq p$, and let v be a leaf in a subtree rooted at vertex u_y, $p < y \leq r$. Since we match $\min\{|Z_1| - 1, \lceil \frac{l}{2} \rceil - a_1\}$ leaves in Z_1 with an equal number of leaves in Z_2 and $|Z_1| \leq |Z_2|$ (Corollary 3.4.14), there is at least one leaf in a subtree rooted at a u_i, $1 \leq i \leq p$, that is not matched and there is also another leaf in a subtree rooted at a u_j, $p < j \leq r$, that is not matched if $l > 3$. The path P contains the root. If the degree of the root is at least 4, u and v satisfy the leaf-connecting condition (Definition 3.2.3). If the degree of the root is 3, P contains either u_i or u_j. The degrees of u_i and u_j are at least 3. Otherwise, P contains both u_i and u_j. Thus u and v satisfy the leaf-connecting condition (Definition 3.2.3). □

Corollary 3.4.17 *Let k be the number of matched pairs found in function* `case2_1`. *Let G' be the resulting graph obtained from the current graph G by adding a new edge between each matched pair of leaves. The value of the lower bound given in Theorem 3.4.1 applied to G' is k less than the value of the same lower bound applied to G, and $blk(G')$ remains balanced. Let l be the number of leaves in $blk(G)$. The number of leaves in $blk(G')$ is at most $\frac{l}{2}$, if $l > 3$.*

Proof. From Claim 3.4.16, the number of leaves in $blk(G')$ is $2k$ less than the number of leaves in $blk(G)$. Since $k \geq \frac{l}{4}$ if $l > 3$ (Corollary 3.4.15), the number of leaves in $blk(G')$ is at most $\frac{l}{2}$. From Corollary 3.4.15, we add at most $\lceil \frac{l}{2} \rceil - a_1$

edges, thus no c-vertex in $blk(G')$ becomes massive. By Lemma 3.4.7, we know that the value of the lower bound given in Theorem 3.4.1 applied to G' is k less than the value of the same lower bound applied to G, and $blk(G')$ remains balanced. □

Subcase 2.2: u^* is a c-vertex Recall that the u_i, $1 \leq i \leq r$, and $x_i \geq x_{i+1}$, $1 \leq i < r$.

We partition the set of subtrees rooted at children of the root into two sets such that we can match leaves between two sets. We first give a claim to show how to partition the set of subtrees.

Claim 3.4.18 *Let q be the largest integer with $x_q \geq 2$. There exists an integer p such that $1 \leq p \leq q$ and $\frac{l}{2} \geq \sum_{i=1}^{p} x_i > \frac{l}{8} + (p-1)$.*

Proof. If $x_1 > \frac{l}{8}$, then $p = 1$. If $x_1 \leq \frac{l}{8}$, we can find an integer p such that $\frac{l}{2} \geq \sum_{i=1}^{p} x_i > \frac{3l}{8}$ using an argument similar to the one given in the proof of Claim 3.4.13. By definition, we know that $x_p \geq 2$ because otherwise the root (a c-vertex) is massive. Thus $p \leq \frac{l}{4}$. Hence $(\sum_{i=1}^{p} x_i) - (p-1) > \frac{l}{8}$. □

Let T_{u_i} be the subtree rooted at u_i. We define the *merge* operation for the collection of subtrees T_{u_i}, $1 \leq i \leq p$, as follows. We first connect the rightmost leaf of T_{u_i} and the leftmost leaf of $T_{u_{i+1}}$, $1 \leq i < p$. This can be done by the fact that each T_{u_i}, $1 \leq i \leq p$, has at least 2 leaves.

Claim 3.4.19 *Let T^* be the block tree obtained from T by collapsing b-vertices that are in the same fundamental cycle created by the addition of new edges introduced by the merge operation.*

(1) The merge operation creates only one b-vertex b^*.

(2) Vertex b^* is a child of the root and b^* is the root of the subtree that contains the updated portion of the block tree.

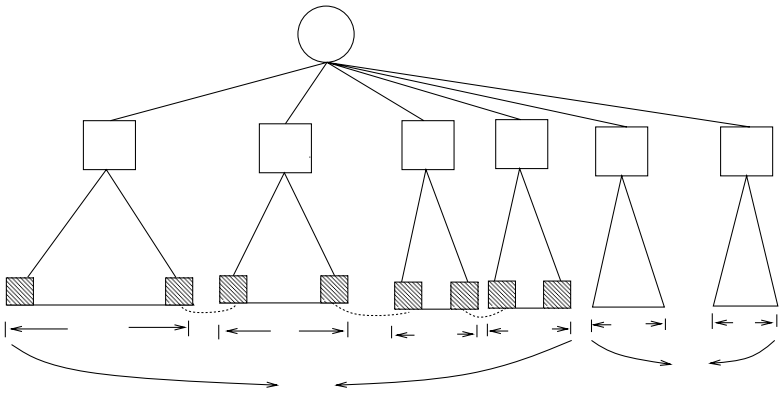


Figure 3.11: The notations used in case 2.2. The number of leaves in the subtree rooted at u_i is x_i. We find the largest p such that the total number of leaves in the first p subtrees rooted at children of the root is greater than $\frac{l}{8}+(p-1)$, but at most $\frac{l}{2}$. We first merge subtrees rooted at u_i, $1 \leq i \leq p$, by connecting the rightmost leaf in the subtree rooted at u_i and the leftmost leaf in the subtree rooted at u_{i+1}, $1 \leq i < p$. Leaves in the first p subtrees rooted at children of u^* are in Y_1. Y_2 consists of the rest of leaves in the tree. We then match $\min\{(\sum_{i=1}^{p} x_i)-1, \lceil \frac{l}{2} \rceil - a_1\} - (p-1)$ leaves in Y_1 with leaves in Y_2.

Proof. Let C_i, $1 \leq i < p$, be the fundamental cycle created by connecting the rightmost leaf of T_{u_i} and the leftmost leaf of $T_{u_{i+1}}$. The cycles C_i and C_{i+1}, $1 \leq i < p-1$, share the b-vertex u_i. From part (2) in Fact 3.3.6, we know that all b-vertices in cycles C_i, $1 \leq i < p$, shrink into a single b-vertex in the new block tree. Let this new b-vertex be b^*. Thus part (1) of the claim holds. Part (2) of the claim follows from part (4) in Fact 3.3.6. □

Note that if we root the updated block tree T^* given in Claim 3.4.19 at the b-vertex b^*, the situation is similar to that in case 2.1. Thus we can match an additional $\min\{(\sum_{i=1}^{p} x_i)-1, \lceil \frac{l}{2} \rceil - a_1\} - (p-1)$ pairs of vertices by pairing up unmatched leaves in subtrees T_{u_i}, $\forall i$, $1 \leq i \leq p$, and leaves in subtrees in subtrees T_{u_i}, $\forall i$, $p < i \leq r$. This algorithm is given in Algorithm 3.8. The notations used are shown in Figure 3.11.

```
set of pairs of vertices function case2_2(tree T);
    vertex u, v; integer p; set of vertices Y1, Y2; set of pairs of vertices L;
    let u_i, 1 ≤ i ≤ r, be the children of the root u*;
    let T_{u_i} be the subtree rooted at u_i; let x_i be the number of leaves in T_{u_i};
    find the largest integer p such that l/2 ≥ Σ_{i=1}^{p} x_i > l/8 + (p − 1);
    Y1 := the set of leaves in the subtrees rooted at u_i, 1 ≤ i ≤ p;
    Y2 := the set of leaves in the subtrees rooted at u_i, i > p;
    L := ∅; {* L is the set of matched pairs. *}
    pfor i = 1 .. p − 1 do
        let u be the leftmost leaf of T_{u_i}; let v be the rightmost leaf of T_{u_{i+1}};
        L := L ∪ {(u, v)}; remove u and v from Y1
    rofp;
    number the leaves in Y1 in arbitrary order from 1 to |Y1|;
    number the leaves in Y2 in arbitrary order from 1 to |Y2|;
    k := min{Σ_{i=1}^{p} x_i, ⌈l/2⌉ − a1} − (p − 1);
    pfor i = 1 .. k do
        u, v := the ith vertex in Y1 and Y2, respectively;
        L := L ∪ {(u, v)}
    rofp;
    return L
end case2_2;
```

Algorithm 3.8: Parallel algorithm for case 2.2.

Corollary 3.4.20 *The number of matched pairs k in case 2.2 satisfies* $\lceil \frac{l}{2} \rceil - a_1 \geq k \geq \frac{l}{8}$, *if* $l > 3$. □

Claim 3.4.21 *Each pair of vertices matched in function* `case2_2` *satisfies the leaf-connecting condition (Definition 3.2.3), if* $l > 3$. □

Corollary 3.4.22 *Let k be the number of matched pairs found in function* `case2_2`. *Let G′ be the resulting graph obtained from the current graph G by adding a new edge between each matched pair of leaves. The value of the lower bound given in Theorem 3.4.1 applied to G′ is k less than the value of the same lower bound applied to G, and blk(G′) remains balanced. Let l be the number of leaves in blk(G). The number of leaves in blk(G′) is at most* $\frac{3l}{4}$, *if* $l > 3$.

Proof. From Claim 3.4.21, the number of leaves in $blk(G')$ is $2k$ less than the number of leaves in $blk(G)$. Since $k \geq \frac{l}{8}$ if $l > 3$ (Corollary 3.4.20), the number

```
set of pairs of vertices function case2(tree T);
{* l is the number of leaves in T; a_1 + 1 is the largest degree of all c-vertices in T. *}
    vertex u*;
    root T at an arbitrary vertex;
    find a vertex u* such that there are more than l/2 leaves in the subtree rooted at u*,
    but none of the subtrees rooted at a child of u* have more than l/2 leaves;
    root T at u*;
    permute the children of u* (from left to right) in non-increasing order of
    the number of leaves in subtrees rooted at them;
    if u* is an b-vertex then return case2_1(T)
    else {* u* is a c-vertex *} return case2_2(T) fi
end case2;
```

Algorithm 3.9: Parallel algorithm for case 2.

of leaves in $blk(G')$ is at most $\frac{3l}{4}$. From Corollary 3.4.20, we add at most $\lceil \frac{l}{2} \rceil - a_1$ edges, thus no c-vertex in $blk(G')$ becomes massive. By Lemma 3.4.7, we know that the value of the lower bound given in Theorem 3.4.1 applied to G' is k less than the value of the same lower bound applied to G, and $blk(G')$ remains balanced. □

The complete algorithm for case 2 is shown in Algorithm 3.9. The correctness of this algorithm is shown earlier in the two subcases (Corollary 3.4.17 and Corollary 3.4.22).

3.5 The Complete Parallel Algorithm and Its Implementation

We first describe the overall parallel algorithm and then an efficient parallel implementation on an EREW PRAM.

3.5.1 The Complete Parallel Algorithm

We present the complete parallel algorithm for the biconnectivity augmentation problem in Algorithm 3.10. The correctness of algorithm `par_bca` follows from the correctness we established earlier for the various cases (Corol-

lary 3.4.11, Corollary 3.4.17 and Corollary 3.4.22). In the previous sections, we have shown details of each step in algorithm `par_bca` except step 1. We now describe an algorithm for updating the block tree given the original block tree T and the set of edges S added to it (step 1 in algorithm `par_bca`).

To describe the parallel algorithm for updating the block graph T after adding a set of edges S, we define the following equivalence relation $\mathcal{R}$ on the set of b-vertices B, where B=$\{v \mid v$ is a b-vertex in T and v is in a cycle created by adding the edges in $S\}$. A pair (x, y) is in $\mathcal{R}$ if and only if $x \in B$, $y \in B$, and vertices in blocks represented by x and y are in the same block after adding the edges in S. It is obvious that $\mathcal{R}$ is reflexive, symmetric, and transitive. Since $\mathcal{R}$ is an equivalence relation, we can partition B into k disjoint subsets B_i, $1 \leq i \leq k$, such that for each i, $x, y \in B_i$ implies $(x, y) \in \mathcal{R}$ and for any $(x, y) \in \mathcal{R}$, x and y both belong to the same B_i.

The following claim can easily be verified by using Fact 3.3.6 and the above definition for the equivalence relation on the set of b-vertices.

Claim 3.5.1 *Two b-vertices b_1 and b_2 are in the same equivalence class if and only if there exists a set of fundamental cycles $\{C_0, \ldots, C_q\}$ such that $b_1 \in C_0$, $b_2 \in C_q$ and C_i and C_{i+1} share a common b-vertex, for $0 \leq i < q$.* □

Notice that fundamental cycles in the block tree created by adding edges between pairs of leaves found in phase 1 and phase 2 of case 1 and subcase 2.1 share a common b-vertex (the root). Any pair of fundamental cycles created by adding edges between pairs of leaves found in phase 3 of case 1 either share a child of v_1 (a b-vertex) or do not share any b-vertex at all. Fundamental cycles created by adding edges between pairs of leaves found in phase 4 of case 1 do not share any b-vertex with any other fundamental cycle. Any pair of

```
graph function par_bca(graph G);
{* The input graph G has at least 3 vertices;
l is the number of leaves in the block graph T. *}
    set of pairs of vertices L; tree T; vertex u, v, x1, x2; set of edges S;
    T := blk(G);
    if T is a forest then perform the procedure specified in Section 3.4.1 fi;
    if T is not balanced then perform the procedure specified in Section 3.4.2 fi;
    while l ≥ 2 do
      if l > 3 then
        if a1 > l/4 then L := case1(T) else {* a1 ≤ l/4 *} L := case2(T) fi
      else {* l ≤ 3 *} let u and v be two leaves in T; L := {(u, v)}
      fi;
      S := ∅;
      pfor each (u, v) ∈ L do
        x1 := a non-cutpoint vertex in the corresponding block of G represented
        by u;
        x2 := a non-cutpoint vertex in the corresponding block of G represented
        by v;
        add an edge between x1 and x2; S := S ∪ {(u, v)}
      rofp;
1.    T := par_update(T, S) {* The procedure par_update returns the updated
      block tree after adding the set of edges in S. *}
    od;
    return G
end par_bca;
```

Algorithm 3.10: Parallel algorithm for finding a smallest augmentation to biconnect a graph.

fundamental cycles created by adding edges between pairs of leaves found in subcase 2.2 share either the root (a b-vertex) or a b-vertex created by the merge operation (Claim 3.4.19).

From the above discussion, we know that b-vertices in fundamental cycles formed by adding edges due to phase 1 and phase 2 of case 1 shrink into a single b-vertex in the new block tree. Let Γ be the fundamental cycles formed by adding edges due to phase 3 of case 1. The b-vertices in Γ which share a common child of v_1 shrink into a single b-vertex. The b-vertices in a fundamental cycle formed by adding edges due to phase 4 of case 1 shrink into a single b-vertex. The b-vertices in all fundamental cycles formed by adding

edges due to subcase 2.1 or subcase 2.2 shrink into a single b-vertex. Thus we know how to compute the equivalence classes of $\mathcal{R}$.

In Algorithm 3.11, we describe our method to update the block tree given the original block tree T and the set of edges S added to it.

Claim 3.5.2 *Function* `par_update` *returns the updated block tree.*

Proof. From Claim 3.5.1 and parts (3) and (4) in Fact 3.3.6. □

Note that we can get the updated block tree by using an algorithm for finding biconnected components. We will, however, show in Section 3.5.2 that the time needed on an EREW PRAM for updating the block tree using function `par_update` is less than what is needed to compute connected components using a linear number of processors. Hence we do not want to use the straightforward algorithm for finding connected components to implement function `par_update`.

3.5.2 The Parallel Implementation

We now describe an efficient parallel implementation for algorithm `par_bca`. Given an undirected graph, we can find its block graph in $O(\log^2 n)$ time using a linear number of processors on an EREW PRAM by the parallel algorithm in Tarjan and Vishkin [TV85] for finding biconnected components and using some procedures in Nath and Maheshwari [NM82].

The parallel versions of stage 1 and stage 2 are described in Section 3.4.1 and Section 3.4.2, respectively. In stage 3, the children-permutation procedure can be done in time $O(\log n)$ using a linear number of processors on an EREW PRAM by calling the parallel merge sort routine in Cole [Col88] and using the Euler tour technique in Tarjan and Vishkin [TV85] to restructure

```
tree function par_update(tree T, set of edges S);
    vertex w; integer k; set of edges S1, S2, S3, S4;
    let B be the set of b-vertices in a cycle in T ∪ S;
    {* The partition {Bi | 1 ≤ i ≤ k} of B is computed such that two b-vertices
    b1 and b2 are in the same set if and only if there exists a set of
    fundamental cycles {C0, ..., Cq} in T ∪ S with b1 ∈ C0, b2 ∈ Cq and Ci and Ci+1
    share a common b-vertex, 0 ≤ i < q. *}
    if S is constructed from pairs found in case 1 then
        Si := the edges in S corresponding to the pairs found in phase i, 1 ≤ i ≤ 4;
        B1 := the set of b-vertices in fundamental cycles in T ∪ S1 ∪ S2;
        pfor the ith child zi of v1 do
            Bi+1 := the set of b-vertices in fundamental cycles in T ∪ S3 that contain zi
        rofp;
        k := 1 + the number of children of v1 in T;
        pfor the ith edge ei in S4 do
            let Bi+k be the set of b-vertices in the fundamental cycle in T ∪ {ei}
        rofp;
        k := k + |S4|
    else {* S is constructed from the pairs found in case 2 *}
        B1 := B; k := 1
    fi;
    pfor i = 1 .. k do
        collapse all b-vertices in Bi into a single b-vertex
    rofp;
    eliminate parallel edges created by collapsing b-vertices;
    let T' be this graph;
    pfor each c-vertex w in T' do if degree(w) = 1 then eliminate w fi rofp;
    return T'
end par_update;
```

Algorithm 3.11: Parallel algorithm for updating the block tree.

and normalize the tree. To perform functions `case1`, `case2`, and `par_update`, we need the following procedures.

- A procedure that numbers leaves in the tree from left to right or from right to left.
- For each vertex v in a tree, find the number and the set of leaves in the subtree rooted at v.
- For a vertex v in a tree, find the leftmost leaf of each subtree rooted at a child of v.

- For a tree T with a set of edges S added between leaves in T, compute:
  - the number of cycles that pass through a vertex in $T \cup S$;
  - the set of vertices in a cycle in $T \cup S$.

All of these procedures can be done in $O(\log n)$ time using a linear number of processors on an EREW PRAM by using the Euler technique in Tarjan and Vishkin [TV85] and certain procedures of Schieber and Vishkin [SV88].

From Corollaries 3.4.11, 3.4.17, and 3.4.22, we know that algorithm `par_bca` removes at least a quarter of the leaves in the current block graph during each execution of the **while** loop. Initially, the number of leaves is at most n. Hence the main **while** loop in algorithm `par_bca` is executed $O(\log n)$ times. Each iteration takes $O(\log n)$ time using a linear number of processors, since the parallel sorting routine used in permuting children needs $O(n)$ processors. This establishes the following claim.

Claim 3.5.3 *The biconnectivity augmentation problem on an undirected graph can be solved in time $O(\log^2 n)$ using a linear number of processors on an EREW PRAM, where n is the number vertices in the input graph.* □

3.6 Concluding Remarks

In this chapter we have presented a linear time sequential algorithm and an efficient parallel algorithm to find a smallest augmentation to biconnect a graph. Our sequential algorithm corrects an error in an earlier algorithm proposed for this problem in Rosenthal and Goldner [RG77]. Our parallel algorithm is new, and it runs in $O(\log^2 n)$ time using a linear number of processors on an EREW PRAM. Although the parallel algorithm follows the overall

structure of our sequential algorithm, the parallelization of some of the steps required new insights into the problem. Our parallel algorithm can be made to run within the same time bound using $O(\frac{(n+m)\log\log n}{\log n})$ processors by using the algorithm for finding connected components in [CV86], and the algorithm for integer sorting in [Hag87].

Chapter 4

Smallest Triconnectivity Augmentation: Biconnected Graphs

4.1 Introduction

In this chapter, we present a linear time sequential algorithm for finding a smallest augmentation to triconnect a biconnected graph. Our sequential algorithm has a similar structure to the one used in biconnectivity augmentation as described in Chapter 3. (We have been informed that Jordán [Jor93b] independently obtained a linear time algorithm for this problem.) We also give an EREW parallel algorithm for solving this problem in $O(\log^2 n)$ time using a linear number of processors. The approach used for the parallel algorithm is similar to the one described in Chapter 3 for finding a smallest biconnectivity augmentation. In Chapter 5, we will consider the problem of finding a smallest triconnectivity augmentation on a graph that is not biconnected. An extended abstract of part of the work presented in this chapter appears in [HR91a].

4.2 Definitions

We first give some definitions.

Higher Connectivity

A graph G is *k-connected*, $k \geq 2$, if and only if G has at least $k+1$ vertices and the removal of any set of vertices of size less than k does not disconnect it. Let $\mathcal{S}$ be a minimal set of vertices whose removal disconnects a k-connected graph G. The set of vertices $\mathcal{S}$ is a *separating k-set*. If $|\mathcal{S}| = 2$, it is a *separating pair*.

Another characterization of k-connected graphs is due to Menger [Men27, Eve79]. A graph G is k-connected, $k \geq 2$, if G contains more than k vertices and there are k vertex-disjoint paths between every pair of vertices in G. The above two definitions are equivalent.

Bridge

Let Q be a subgraph of a graph G. We define the *bridges* of Q in G as follows: we partition vertices in $G - Q$ into classes such that two vertices are in the same class if and only if there is a path connecting them which does not use any vertex of Q. Each such class K defines a *nontrivial bridge* $B = (V_B, E_B)$ of Q, where B is the subgraph of G with $V_B = K \cup \{\text{vertices of } Q \text{ that are connected by an edge to a vertex in } K\}$, and E_B containing the edges of G incident on a vertex in K. An edge in $G - Q$ with both endpoints in Q is a *trivial bridge* of Q. The nontrivial and trivial bridges of Q together form the *bridges* of Q.

Tutte Split

Let $\{a_1, a_2\}$ be a separating pair in G and let G_a be the subgraph of G induced on $\{a_1, a_2\}$. For any bridge X of G_a, let V_X be the set of vertices in X; let $\overline{X}$ be the induced subgraph of G on $(V - V_X) \cup \{a_1, a_2\}$. Let B be a bridge of G_a such that B and $\overline{B}$ both contain at least two edges and either B or $\overline{B}$ is biconnected. The *Tutte split* operation on $\{a_1, a_2\}$ defined in Tutte [Tut66] (see also Ramachandran [Ram93]) on a biconnected graph G forms two graphs $G_1 = B \cup \{(a_1, a_2)\}$ and $G_2 = \overline{B} \cup \{(a_1, a_2)\}$. The edge (a_1, a_2) added in G_1 and G_2 is called a *virtual edge*.

Tutte Component

Let G' be the graph obtained from a biconnected graph G by performing the Tutte split operation successively until no Tutte split is possible. An example

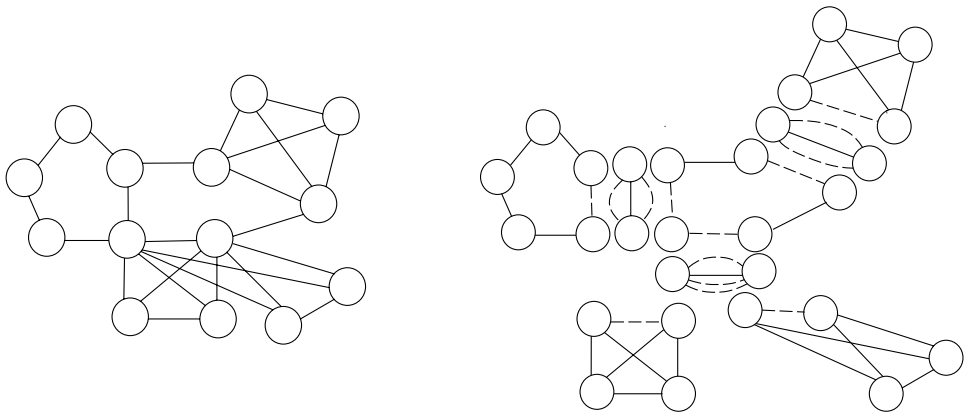


Figure 4.1: A biconnected graph G and the graph obtained from G by repeated application of Tutte split. Virtual edges created by Tutte splits are shown as dashed lines. Note that several vertices are duplicated after applying Tutte splits.

is shown in Figure 4.1. Every connected component in G' is called a *Tutte component*. From [Tut66], we know that every Tutte component is of one of the following three types: (i) a triconnected component; (ii) a simple cycle (a polygon); (iii) a pair of vertices with at least three edges between them (a bond).

3-Block Graph

Given a biconnected graph G, we define the *3-block graph*, 3-$blk(G)$, as follows. (This is essentially the tree of triconnected components [HT73] with a few variations.) Let G' be the graph obtained from a biconnected graph G by performing the Tutte split operation successively until no Tutte split is possible. The 3-block graph contains three sets of vertices: β-vertices, σ-vertices, and π-vertices. For every Tutte component that is a triconnected component in G', we create a β-vertex in 3-$blk(G)$. For every polygon Q in G', we create a π-vertex; if w is a vertex in Q with degree 2 in G, we create a β-vertex b_w for w, and we call w the corresponding Tutte component represented by b_w. A

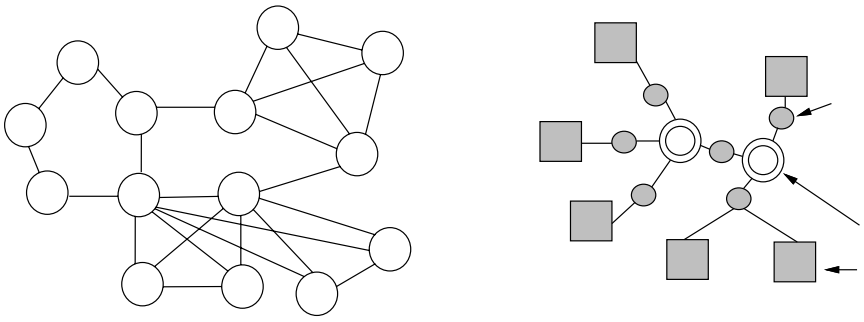


Figure 4.2: A graph G and 3-$blk(G)$. We represent a σ-vertex, a β-vertex and a π-vertex by a shadowed circle, a shadowed rectangle and two concentric circles, respectively. The vertex-numbers appearing with each σ-vertex in 3-$blk(G)$ represent the corresponding Tutte pair in G. The vertex-numbers appearing with each β-vertex in 3-$blk(G)$ represent the vertices in its corresponding Tutte component.

β-vertex that corresponds to a single vertex in G is a *trivial β-vertex*.

Let z_1 and z_2 be the two vertices in Q that are adjacent to a degree-2 vertex w. We create a σ-vertex for the pair of vertices z_1 and z_2. For every pair of vertices a_1 and a_2 in G, we create a σ-vertex for (a_1, a_2) if we have performed a Tutte split with respect to a_1 and a_2 in G. Examples of a σ-vertex, a β-vertex, and a π-vertex are shown in Figure 4.2, where they are represented by a shadowed circle, a shadowed rectangle, and two concentric circles, respectively.

Vertices u and v in 3-$blk(G)$ are adjacent if any of the following conditions holds: (i) u is a β-vertex corresponding to a Tutte component H_u that is a triconnected component, v is a σ-vertex, and H_u contains the pair of vertices corresponding to v; (ii) u is a β-vertex corresponding to a degree-2 vertex w in G and v is the σ-vertex corresponding to the pair of vertices in G that are adjacent to w; (iii) u is a π-vertex corresponding to a polygon Q in H and v is σ-vertex corresponding to a pair of vertices in Q; (iv) interchanging u and v in any one of the previous three conditions.

The pairs of vertices that correspond to σ-vertices are *Tutte pairs*. It is easy to see that a Tutte pair must be a separating pair while the reverse is not necessarily true. Further, it is well-known that a biconnected graph with n vertices can have $\Theta(n^2)$ separating pairs while the number of Tutte pairs in an n-vertex biconnected graph is $O(n)$ [Kan88, Ram93]. Figure 4.2 shows the 3-block graph of the original graph whose Tutte components are shown in Figure 4.1.

From [HT73, Ram93, Tut66], we know that 3-$blk(G)$ is a tree if G is biconnected. We call this tree the *3-block tree* for G. Each Tutte component that corresponds to a β-vertex in the 3-block graph is a *3-block* of G. Given a biconnected graph G with n vertices and m edges, 3-$blk(G)$ can be constructed in $O(n+m)$ time using procedures in [HT73, FRT93, Ram93]. The 3-block tree can also be constructed in $O(\log n)$ time using a linear number of processors on a CRCW PRAM [FRT93, Ram93]. The algorithm for constructing the 3-block tree can be made to run within the same time bound using a sublinear number of processors by using the algorithm for finding connected components in [CV86] and the algorithm for integer sorting in [Hag87].

Implied Path in the 3-Block Tree

Given two vertices u and v in a biconnected graph G, the *implied path* between u and v in 3-$blk(G)$ is the path between the two β-vertices that corresponds to the two Tutte components that contain u and v, respectively. An example of an implied path in 3-$blk(G)$ is shown in Figure 4.3.

Degrees for σ-Vertices and Tutte Pairs

Let G be a biconnected graph. Given a σ-vertex s in 3-$blk(G)$, let (a_1, a_2) be its corresponding Tutte pair. We define $d_3((a_1, a_2))$ or $d_3(s)$ to be the degree of s in 3-$blk(G)$. Note that $d_3((a_1, a_2))$ is greater than or equal to 2 for any

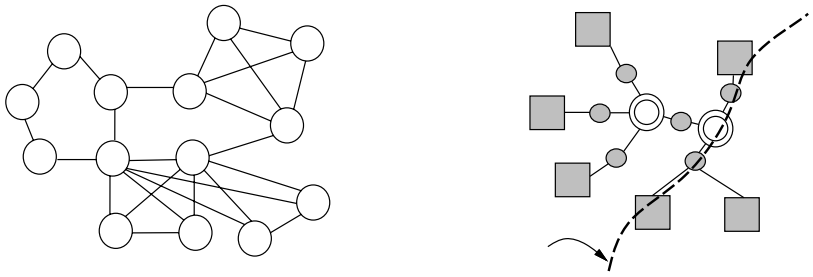


Figure 4.3: The graph G, 3-$blk(G)$, and the implied path P between vertices 6 and 12 in 3-$blk(G)$.

Tutte pair (a_1, a_2). Note also that the degree for a separating pair s is equal to the number of components in the graph obtained from G by removing s.

Chain

Given a σ-vertex s in 3-$blk(G)$, let (3-$blk(G) - s$) be the graph obtained from 3-$blk(G)$ by removing s. An *s-chain* [RG77] is a component of 3-$blk(G) - s$ which contains only one 3-block leaf in 3-$blk(G)$. The 3-block leaf of 3-$blk(G)$ in an s-chain is called the *s-chain leaf*.

Triconnectivity Augmentation Number

Given a graph G, the *triconnectivity augmentation number* of G, $A_3(G)$, is the smallest number of edges that need to be added to G to triconnect G.

4.3 A Lower Bound for the Triconnectivity Augmentation Number

In this section, we identify disjoint portions of the input graph that must have a new incoming edge in any triconnectivity augmentation. Based on this information, we give a lower bound for the triconnectivity augmentation number that is similar to the one given for the biconnectivity augmentation number in [ET76].

We first identify β-vertices in a 3-block graph whose corresponding Tutte components must contain a new incoming edge in order to triconnect the input graph.

Definition 4.3.1 *A degree-1 β-vertex in the 3-block tree of a biconnected graph is a* **3-block leaf**.

Let H be a Tutte component that corresponds to a 3-block leaf in 3-$blk(G)$. From Definition 4.3.1, we know that G is disconnected if we remove the Tutte pair in H from G. If we want to triconnect the input graph, we must add at least one new incoming edge to H.

We now identify vertices in H such that if we add an edge between one of the vertices we identified and a vertex not in H, then H no longer corresponds to a 3-block leaf. These vertices will be used in the development of the algorithm for finding a smallest triconnectivity augmentation (Section 4.4.2).

Definition 4.3.2 *Given a 3-block leaf b in 3-$blk(G)$, let H_b be its corresponding Tutte component. A vertex u in H_b is a* **demanding vertex** *of b if u is not in the Tutte pair of G that is contained in H_b. The vertex u is also called a demanding vertex in G.*

Claim 4.3.3 *There exists at least one demanding vertex in every 3-block leaf in the 3-block tree of a biconnected graph.* □

When we specify the 3-block graph of a graph G, we will include a demanding vertex for each 3-block leaf in the 3-block graph.

We now define the *weight* of a graph, which we will relate later to its triconnectivity augmentation number.

Definition 4.3.4 *Let G be a biconnected graph. Let $l_3(G)$ be the number of 3-block leaves in 3-blk(G). The* **weight** *of the graph G, $w(G)$, is $l_3(G)$.*

We now state a lower bound for the triconnectivity augmentation number of a biconnected graph. The proof of this lemma is similar to a proof given in [ET76].

Lemma 4.3.5 *We need at least* $\max\{d-1, \lceil \frac{w(G)}{2} \rceil\}$ *edges to triconnect a biconnected graph G, where d is the largest degree among all σ-vertices in 3-blk(G).* □

4.4 Finding a Smallest Augmentation to Triconnect a Biconnected Graph

In this section, we consider the problem of finding a smallest set of edges to triconnect a biconnected graph. We show that the lower bound given in Lemma 4.3.5 can be achieved, and we give a linear time algorithm to find such a smallest triconnectivity augmentation.

4.4.1 Properties of the 3-Block Tree for a Biconnected Graph

In this section, we explore properties of 3-$blk(G)$ that will be used in the following sections.

Massive, Critical, and Balanced

For a biconnected graph G with weight $w(G)$, a Tutte pair s or its corresponding σ-vertex is *massive* if $d_3(s) - 1 > \lceil \frac{w(G)}{2} \rceil$. A Tutte pair s or its corresponding σ-vertex is *critical* if $d_3(s) - 1 = \lceil \frac{w(G)}{2} \rceil$. If no Tutte pair in G is *massive*, then G and its 3-block graph are called *balanced*. These definitions are analogous to the same types of vertices for biconnectivity augmentation as defined in Chapter 3.

First we give bounds on the number of massive and critical σ-vertices in 3-$blk(G)$. They are similar to the bounds on the number of massive and critical c-vertices in the 2-block graph given in Chapter 3. Proofs given in Chapter 3 can be easily modified to prove the following claim and its corollary.

Claim 4.4.1 *Let s_1 be a σ-vertex with the largest degree among all σ-vertices in 3-$blk(G)$, and let s_i be a σ-vertex with the largest degree among all σ-vertices other than $s_1, \ldots, s_{i-1}$, for $i \in \{2, 3\}$. Then $\sum_{i=1}^{3} d_3(s_i) - 4 \leq l$, where l is the number of 3-block leaves in 3-$blk(G)$.* □

Corollary 4.4.2 *Let s_1, s_2, and s_3 be the σ-vertices defined in Claim 4.4.1 and let l be the number of 3-block leaves in 3-$blk(G)$.*
(i) If 3-$blk(G)$ has more than two σ-vertices, then $d_3(s_3) \leq \frac{l+4}{3}$.
(ii) There can be at most one massive σ-vertex in 3-$blk(G)$.
(iii) If there is a massive σ-vertex in 3-$blk(G)$, then there is no critical σ-vertex in 3-$blk(G)$.
(iv) There can be at most two critical σ-vertices in 3-$blk(G)$ if $l > 2$. □

Given a biconnected graph G, its 3-block tree, and a graph G' obtained from G by adding an edge between two distinct vertices u and v, we now describe a method to obtain 3-$blk(G')$ from 3-$blk(G)$ by local updating operations on 3-$blk(G)$ instead of computing it directly from G'. Let u and v be two vertices of G that are in Tutte components represented by β-vertices t_u and t_v, respectively. Let P be the implied path between u and v in 3-$blk(G)$.

The Implied Graph

We give some more notations. Given G, 3-$blk(G)$, G', and P as defined in the previous paragraph, we define the *implied graph G_P of the path P* in G

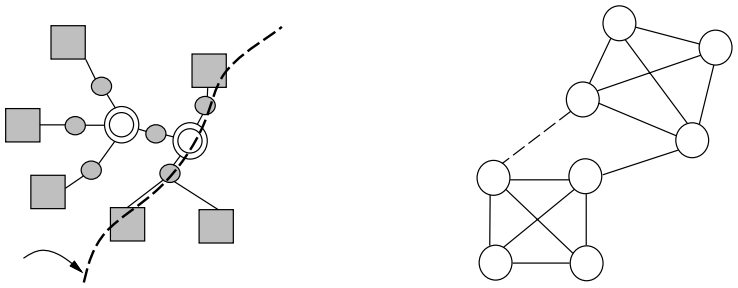


Figure 4.4: The implied path P between vertices 6 and 12 in 3-$blk(G)$ and its implied graph. The original graph G is shown in Figure 4.3.

as follows. The implied graph G_P contains vertices and edges that are in the Tutte components of G corresponding to β-vertices in P. For every π-vertex q in P, let s_a^q and s_b^q be the two σ-vertices that are adjacent to q in P. Let (a_1^q, a_2^q) and (b_1^q, b_2^q) be the Tutte pairs represented by s_a^q and s_b^q, respectively. If s_a^q is adjacent to a trivial β-vertex in P that corresponds to the vertex w in G, then let $a_1^q = w$ and let $a_2^q = w$. The same procedure is applied on s_b^q, b_1^q, and b_2^q. If we traverse clockwise around the simple cycle C_q represented by q starting from a_1^q, let the sequence of vertices visited in $\{a_2^q, b_1^q, b_2^q\}$ is b_1^q, b_2^q and a_2^q. We add the edge (a_1^q, b_1^q) to G_P if $a_1^q \neq b_1^q$ and we add the edge (a_2^q, b_2^q) to G_P if $a_2^q \neq b_2^q$. An example is shown in Figure 4.4.

The implied graph for any path P has some very useful properties. The following claim shows that the implied graph of a path P is biconnected if G_P contains at least three vertices. Note that G_P contains a single vertex if and only if $u = v$. The implied graph G_P contains two vertices with two edges between them if and only if u and v are both degree-2 in G and u and v are in the same polygon. Further, we will show later (in part *ii* of Lemma 4.4.5) that the implied graph becomes triconnected after adding an edge between u and v if it contains at least 4 vertices.

Claim 4.4.3 *Given a biconnected graph G and two demanding vertices u and v in G where u and v are in different Tutte components, let P be the implied path in 3-blk(G) between u and v. The implied graph of P, G_P, is biconnected if and only if G_P contains more than two vertices.*

Proof. The only if part is obvious. We now prove the if part.

By the definition of 3-block graph, we know that a Tutte component corresponding to a nontrivial β-vertex is triconnected. Since G_P contains more than two vertices, u and v are not in the same polygon. Thus P contains at least one nontrivial β-vertex. Hence G_P contains at least 4 vertices.

Given any two distinct vertices x_1 and x_2 in G_P, let P' be the implied path between x_1 and x_2 in 3-$blk(G)$. We now show that there are two vertex-disjoint paths between x_1 and x_2 in G_P. If there is only one vertex r in P', then x_1 and x_2 are contained in the Tutte component H_r that corresponds to r. Since u and v are distinct, H_r is nontrivial and is triconnected. Thus there are two vertex-disjoint paths between x_1 and x_2 in G_P.

If P' consists of more than one vertex, let $P' = [t_1, z_1, \ldots, z_r, t_2]$. Note that P' is a subpath of P. Let k be the number of β-vertices and π-vertices in $\{z_1, \ldots, z_r\}$ and let y_i be the ith β-vertex or π-vertex encountered when we traverse P' from z_1 to z_r. Let s_{2i} and s_{2i+1} be the two σ-vertices that are adjacent to y_i, $1 \leq i \leq k$. We construct two vertex-disjoint paths $\cup_{i=0}^{k+1} P_i^1$ and $\cup_{i=0}^{k+1} P_i^2$ between x_1 and x_2 in G_P, where P_0^1 and P_0^2 are two vertex-disjoint paths in G_P from x_1 to the two vertices in the Tutte pair represented by z_1 (z_3 if t_1 is trivial); P_{k+1}^1 and P_{k+1}^2 are two vertex-disjoint paths in G_P from x_2 to the two vertices in the Tutte pair represented by z_r (z_{r-2} if t_2 is trivial); P_i^1 and P_i^2, $1 \leq i \leq k$, are two vertex-disjoint paths from the two vertices in the Tutte pair represented by s_{2i} to the two vertices in the Tutte pair represented

by s_{2i+1}. We choose the numbering of each P_i^1 and P_i^2 in such a way that $P_i^1 \cap P_{i+1}^1 \neq \emptyset$ and $P_i^2 \cap P_{i+1}^2 \neq \emptyset$, $0 \leq i \leq k$.

We have proved that there are two vertex-disjoint paths between any two distinct vertices in G_P, and G_P contains at least 4 vertices. Thus G_P is biconnected. This proves the if part. □

We now define the *crack* operation which is useful in describing the relation between 3-$blk(G)$ and the 3-block graph of G after adding an edge.

Definition 4.4.4 *Let G be a biconnected graph and let G' be the graph obtained from G by adding an edge between two demanding vertices u and v in G, where u and v are in different Tutte components. Given P, the implied path between u and v in 3-$blk(G)$ and a π-vertex q in P, let s_a^q and s_b^q be the two σ-vertices in P that are adjacent to q. Let (a_1^q, a_2^q) and (b_1^q, b_2^q) be the two Tutte pairs in G that correspond to s_a^q and s_b^q, respectively. We assume that if we traverse clockwise around the simple cycle C_q represented by q starting from a_1^q, the sequence of vertices visited in $\{a_2^q, b_1^q, b_2^q\}$ is b_1^q, b_2^q and a_2^q. The* **crack** *operation on q with respect to P consists of the following procedures on G and 3-$blk(G)$.*

(i) In G, we add the edge (a_1^q, b_1^q) to the simple cycle C_q corresponding to q if $a_1^q \neq b_1^q$ and (a_1^q, b_1^q) is not an edge in C_q. The edge (a_2^q, b_2^q) is added under the same condition for a_2^q and b_2^q. For each new simple cycle C created by adding these edges, we create a new π-vertex in 3-$blk(G)$ if C contains a Tutte pair in C_q (excluding (a_1^q, a_2^q) and (b_1^q, b_2^q)).

(ii) After performing operations given in part (i), let q_1 and q_2 be the two π-vertices corresponding to simple cycles containing (a_1^q, b_1^q) and (a_2^q, b_2^q), respectively in 3-$blk(G)$. New σ-vertices s_1 (corresponding to the new Tutte pair (a_1^q, b_1^q)) and s_2 (corresponding to the new Tutte pair (a_2^q, b_2^q)) are added and

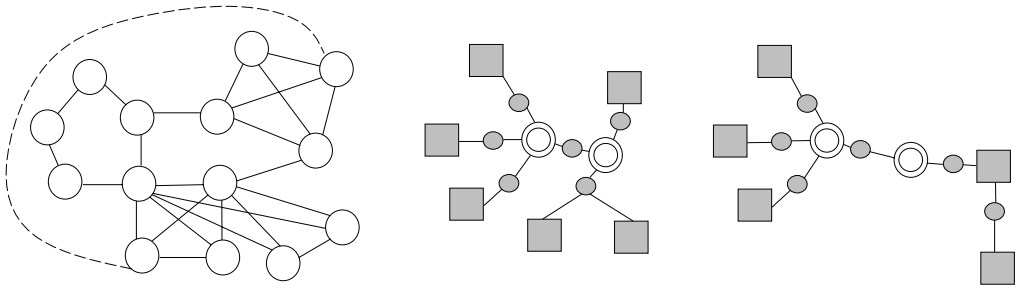


Figure 4.5: A graph G, 3-$blk(G)$, and the updated 3-block graph after adding a new edge between vertices 6 and 12 (represented by a dotted line). The π-vertex q_3 in the updated 3-block graph is created by the crack operation performed on q_2. Note that the implied path between vertices 1 and 8 is adjacent on polygon q_1, while the implied path between vertices 2 and 8 is non-adjacent on polygon q_1.

connected to q_1 and q_2, respectively, in 3-$blk(G)$.

(iii) After performing operations given in parts i and ii, let s be a σ-vertex in 3-$blk(G)$ such that $s \neq s_a^q$ and $s \neq s_b^q$. An edge (s, q) in 3-$blk(G)$ is changed to (s, q_1) or (s, q_2) depending on whether the Tutte pair represented by s is on the simple cycle represented by q_1 or by q_2.

Note that the crack operation creates at most two π-vertices. If there are two π-vertices created by performing the crack operation on a π-vertex q with respect to a path P, then P is *non-adjacent* on q. Otherwise, P is *adjacent* on q. An example is shown in Figure 4.5.

Lemma 4.4.5 states how 3-$blk(G')$ is obtained from 3-$blk(G)$, where G' is the graph obtained form G by adding an edge. (A similar result was found independently in [DBT90].)

Lemma 4.4.5 *Let G be a biconnected graph and let G' be the graph obtained from G by adding an edge between two vertices u and v in G, where u and v are in different Tutte components. Let P be the implied path between u and v*

in 3-blk(G). We can obtain 3-blk(G') from 3-blk(G) by applying the following operations.

(i) Edges in P are eliminated. Vertices and edges in 3-blk(G) that are not in P remain in 3-blk(G').

(ii) The β-vertices in P shrink into a new β-vertex b_P in 3-blk(G'). The corresponding Tutte component of b_P is $G_P \cup \{(u,v)\}$, where G_P is the implied graph of P.

(iii) A vertex s in P with $d_3(s) = 2$ is eliminated in 3-blk(G') if s is a σ-vertex or a π-vertex.

A σ-vertex s in P with $d_3(s) \geq 3$ is adjacent to b_P if b_P exists.

(iv) A π-vertex q in P with $d_3(q) \geq 3$ is cracked (Definition 4.4.4) and new σ-vertices created by the crack operation are connected to b_P.

Proof. Parts (i) and (iii) are obvious. We only prove parts (ii) and (iv).

Since G_P are in different Tutte components, G_P contains at least 2 vertices. If G_P contains exactly two vertices $\{u, v\}$, then $G_P \cup \{(u,v)\}$ is a bond. If G_P contains more than two vertices, G_P is biconnected (Claim 4.4.3). (In the proof of Claim 4.4.3 we also show that G_P must contain at least 4 vertices in this case.) We know that the only way to disconnect G_P is by removing Tutte pairs corresponding to σ-vertices in P. After the removal of any Tutte pair, G_P contains exactly two connected components and u and v are separated. Thus the graph G'_P obtained from G_P by adding (u, v) is triconnected. This proves part (ii).

We know that pairs of endpoints of edges added in cracking π-vertices correspond to new Tutte pairs. Every new Tutte pair created is contained in G'_P. This proves part (iv). □

An example of updating the 3-block graph is shown in Figure 4.5.

Corollary 4.4.6 *Let G' be the graph obtained from G by adding an edge between any two demanding vertices u and v in G and let P be the implied path in 3-blk(G) between u and v. Then the degree of each σ-vertex s in P, with degree > 2 in 3-blk(G), is decreased by 1 in 3-blk(G').* □

In Definition 4.4.7, we specify the conditions under which the addition of an edge between two 3-block leaves in 3-$blk(G)$ reduces the number of 3-block leaves by two. Thus if we add an edge between two vertices satisfying the condition given in Definition 4.4.7, we can reduce the first part of the lower bound (Lemma 4.3.5) by 1.

Definition 4.4.7 (The leaf-connecting condition)
Let G be a biconnected graph and let u and v be two demanding vertices in G. Let P be the implied path between u and v in 3-blk(G). The pair of vertices u and v satisfies the **leaf-connecting condition** *if and only if any one of the following conditions holds:*
(i) the path P contains a β-vertex of degree at least 4;
(ii) the path P contains two vertices of degree at least 3;
(iii) there exists a π-vertex in P such that P is non-adjacent to it.

Lemma 4.4.8 *Let G be a biconnected graph and G' be the graph obtained from G by adding a new edge (u, v). Let l be the number of 3-block leaves in 3-blk(G), and let l' be the number of 3-block leaves in 3-blk(G'). If u and v satisfy the leaf-connecting condition (Definition 4.4.7) and $l > 3$, then $l' = l - 2$.*

Proof. If we substitute β-vertices with b-vertices and σ-vertices with c-vertices, parts (i) and (ii) of the leaf-connecting condition (Definition 4.4.7) are similar to the leaf-connecting condition given in Chapter 3 for biconnecting a graph.

For a proof of the lemma when u and v satisfy part i or part ii of the leaf-connecting condition, see Chapter 3.

Let u and v satisfy part (iii) of the leaf-connecting condition. We know that two 3-block leaves in 3-$blk(G)$ that correspond to Tutte components containing u and v are eliminated in 3-$blk(G')$. Since degree-1 vertices must be β-vertices, we have to prove no new degree-1 β-vertex is created. From Lemma 4.4.5, we know that at least two new π-vertices p_1 and p_2 are created by the crack operation if part (iii) of the leaf-connecting condition is satisfied. The new β-vertex created is adjacent to two σ-vertices that are connected to p_1 and p_2, respectively. Hence it is not a 3-block leaf. □

4.4.2 The Sequential Algorithm

In this section, we describe an algorithm (`aug2to3`) to triconnect a biconnected graph using exactly the number of edges given in Lemma 4.3.5. Given a biconnected graph G, let l be the number of 3-block leaves in 3-$blk(G)$. Let d be the largest degree among all σ-vertices in 3-$blk(G)$. We will show in Claim 4.4.10 that by using Algorithm 4.1, the lower bound given in Lemma 4.3.5 can be reduced by 1 each time we add a new edge.

Algorithm `aug2to3` treats two cases separately. If G is not balanced, there can be only one massive σ-vertex s (Corollary 4.4.2). We add an edge (u, v) such that d and l are both decreased by 1 in the resulting 3-block graph. This can be done by choosing u and v such that the two Tutte components containing u and v are both corresponding to s-chain leaves. If G is balanced, there can be at most two critical σ-vertices (Corollary 4.4.2). We add an edge (u, v) such that d is decreased by 1 and l is decreased by 2. This can be done by choosing u and v such that they satisfy the leaf-connecting condition (Definition 4.4.7),

and the implied path between them in 3-$blk(G)$ passes through all possible critical σ-vertices. In algorithm `aug2to3`, we only consider the case when 3-$blk(G)$ contains a β-vertex with degree ≥ 2. If there is no such β-vertex in 3-$blk(G)$, the 3-block tree either is triconnected, contains two leaves or is a star with a π-vertex as its center. It is easy to augment the graph with the smallest number of edges in all of the above cases.

We describe the method for optimally triconnecting a biconnected graph in Algorithm 4.1. Before we prove the correctness of algorithm `aug2to3`, we state a claim for the case when the input graph G is unbalanced. The proof of this claim is similar to a proof in [RG77] (see Chapter 3).

Claim 4.4.9 *Let G be an unbalanced biconnected graph with at least 4 vertices. Let s be the massive σ-vertex in 3-$blk(G)$ and let $\delta = d_3(s) - 1 - \lceil \frac{l}{2} \rceil$. There are $2\delta + 2$ s-chains in 3-$blk(G)$. Let $\mathcal{M}$ be the set of s-chain leaves. By adding $2k, k \leq \delta$, edges to connect $2k+1$ vertices of $\mathcal{M}$, we reduce both $d_3(s)$ and the number of leaves in the 3-block tree by k.* □

We now prove the correctness of algorithm `aug2to3`.

Claim 4.4.10 *Let d be the largest degree among all σ-vertices in 3-$blk(G)$ and let l be the number of 3-block leaves in 3-$blk(G)$. If G is biconnected with at least 4 vertices, we can triconnect G by adding $\max\{d-1, \lceil \frac{l}{2} \rceil\}$ edges using algorithm* `aug2to3`.

Proof. If 3-$blk(G)$ contains a massive vertex s_1, then the two s_1-chain leaves are found in step 1. The correctness of algorithm `aug2to3` when step 1 is executed is proved in Claim 4.4.9. If 3-$blk(G)$ is balanced, all critical vertices are in $P_1 \cup P_2$. (For details, see [RG77].) Thus d is decreased by 1 after adding

```
{* The input graph G is biconnected with at least 4 vertices; 3-blk(G) contains
a non-leaf β-vertex; the algorithm finds a smallest augmentation to triconnect G. *}
set of pairs of vertices function aug2to3(graph G);
    tree T; vertex v, w, y, z, u1, u2;
    {* Vertices v and w are the two vertices in the 3-block tree that satisfy
    the leaf-connecting condition (Definition 4.4.7), if G is balanced;
    vertices y and z are two degree-1 vertices in the 3-block tree such that the path
    between them in the 3-block tree passes through v and w; vertices u1 and u2
    are demanding vertices of y and z, respectively. *}
    T := 3-blk(G); root T at a non-leaf β-vertex b; S := ∅;
    let l be the number of degree-1 vertices in T;
    while l ≥ 2 do
        let s1 be a σ-vertex with the largest degree in T;
        if s1 is massive then
1.          find two s1-chain leaves y and z
        else {* s1 is not massive *}
            if d3(s1) > 2 then
2.              v := s1
            else if d3(s1) ≤ 2 then
3.              let v be the vertex with the largest degree in T
            fi;
            find the path P1 from v to the root b;
            find a path P2 from b to a leaf v'' such that P2 does not pass v;
            if ∃ a vertex other than v in P1 ∪ P2 with degree ≥ 3 then
4.              w := v''
            else {* all vertices other than v in P1 ∪ P2 are degree-1 or degree-2 *}
5.              find a path P3 from v to a leaf ṽ in the subtree rooted at v such that if v is a
                π-vertex of degree ≥ 4, then P1 ∪ P2 ∪ P3 is non-adjacent on v;
                {* If v is a π-vertex, a path is non-adjacent on v if the crack operation
                (Definition 4.4.4) performed on v produces two new π-vertices. *}
                w := ṽ;
                find two degree-1 vertices y and z such that the path between them
                passes through v and w
            fi
        fi;
        find a demanding vertex u1 of y; find a demanding vertex u2 of z;
        {* Claim 4.3.3 shows that the above two statements will always succeed. *}
        S := S ∪ {(u1, u2)}; update the 3-block graph T;
        if l ≠ 3 then l := l − 2 else {* l = 3 *} l := l − 1 fi
    od;
    return S
end aug2to3;
```

Algorithm 4.1: A linear time algorithm for finding a smallest triconnectivity augmentation on biconnected graph.

an edge (Corollary 4.4.6). The pair of vertices u_1 and u_2 also satisfies part (ii) of the leaf-connecting condition (Definition 4.4.7) by steps 2, 3, and 4, if possible. Otherwise, it satisfies part (i) of the leaf-connecting condition by step 3, or satisfies part (iii) of the leaf-connecting condition by step 5. Thus if $d-1 < \lceil \frac{l}{2} \rceil$, then l is decreased by 2 each time algorithm `aug2to3` adds an edge.

The algorithm guarantees that $\max\{d-1, \lceil \frac{l}{2} \rceil\}$ is decreased by 1 each time we add an edge. We know that $\max\{d-1, \lceil \frac{l}{2} \rceil\} = 0$ implies that 3-$blk(G)$ is a single β-vertex. Thus G is triconnected if G contains at least 4 vertices. Hence the claim holds. □

Theorem 4.4.11 *The triconnectivity augmentation number for a biconnected graph G, $A_3(G)$, equals* $\max\{d-1, \lceil \frac{l}{2} \rceil\}$, *where d is the largest degree among all σ-vertices in 3-$blk(G)$ and l is the number of 3-block leaves in 3-$blk(G)$.*

Proof. By Lemma 4.3.5 and Claim 4.4.10. □

Claim 4.4.12 *Algorithm* `aug2to3` *runs in linear time.*

Proof. The 3-block graph can be obtained in linear time by using procedures in [HT73, Ram93]. If G is a biconnected graph with n vertices and m edges, 3-$blk(G)$ consists of $O(n)$ vertices.

Using a hash table technique that is the same as the one used in [RG77], we can maintain the current degree of all vertices in 3-$blk(G)$. We can also find a vertex in 3-$blk(G)$ with a given degree in constant time using the hash table. The sum of the degrees of all vertices in 3-$blk(G)$ is $O(n)$ and the degree of any vertex in 3-$blk(G)$ can only be decreased by adding edges.

During the entire computation, we create only $O(n)$ vertices and edges in 3-$blk(G)$. Hence it takes $O(n)$ time in total to maintain the hash table.

We use the following data structure to represent the rooted 3-block tree [RG77]. Each vertex in 3-$blk(G)$ has 4 pointers: (i) a left-sibling pointer; (ii) a right-sibling pointer; (iii) a parent pointer; (iv) a child pointer. Sibling nodes form a linear doubly-linked list using the two sibling pointers. The sibling linked list for the children of a π-vertex r will be ordered according to its relative positions in the simple polygon represented by r. The child pointer of a node points to (an arbitrary) one of its children. The parent pointer of a node points to its parent.

We always maintain sibling pointers and the child pointer for each node; however, during the updating of 3-$blk(G)$, the parent pointer will not be changed. To determine the actual parent pointer, we use the property that every edge added collapses β-vertices into the root. The parent pointer of a β-vertex always points to its parent in the current tree. If two π-vertices are created by the crack operation, we can initialize their parent pointers in constant time. If the parent of a σ-vertex was a β-vertex and is collapsed into the root, its current parent is the root. If the parent of a σ-vertex s was a π-vertex and is cracked, then the current parent of s, $parent(s)$, is a grandchild of the root and the current grandparent of s, $grandparent(s)$, is a degree-2 σ-vertex. Our algorithm does not need to access $parent(s)$ and $grandparent(s)$ to perform further computation except that we need to know whether or not the degree of $parent(s)$ is greater than 2. This can be done in constant time by checking the sibling list containing s. Using this data structure, paths P_1, P_2, and P_3 used in algorithm `aug2to3` can be found in $O(|P_1 \cup P_2 \cup P_3|)$ time.

Each vertex and each edge is visited a constant number of times during the entire computation, so the total time for visiting and updating the

tree structure is $O(n)$. Since we create at most one 3-block leaf in the entire computation of algorithm `aug2to3` (when there are three 3-block leaves left), we can find a demanding vertex in each degree-1 vertex in constant time after an $O(n+m)$ preprocessing procedure. Thus the overall time complexity is $O(n+m)$. □

4.5 The Parallel Algorithm

In this section, we develop an efficient parallel algorithm for finding a smallest augmentation to triconnect a biconnected graph.

Given a biconnected graph G, 3-$blk(G)$ is a tree. Our parallel algorithm is similar to the parallel algorithm for finding a smallest biconnectivity augmentation given in Chapter 3. Note that if 3-$blk(G)$ is unbalanced, we can easily add edges in parallel to balance it using a method similar to the one given in Chapter 3. Hence we focus our discussion on the case when the input graph G is balanced.

Let $\{s_1, \ldots, s_{n_s}\}$ be the set of σ-vertices in 3-$blk(G)$ where G is biconnected and balanced. Without lose of generality, let $d_3(s_i) \geq d_3(s_{i+1})$, $1 \leq i < n_s$. Let l denote the number of 3-block leaves in 3-$blk(G)$.

Here is a brief summary of our parallel algorithms. The full details are given in the following subsections. Our algorithm first transforms 3-$blk(G)$ into the following format. We root 3-$blk(G)$ at a vertex such that s_1 is the left child of the root. We further permute the children of s_1 such that the number of leaves in the subtrees rooted at the children of s_1 is non-increasing from left to right. Let $U_i = \{u \mid u \text{ is the leftmost leaf of } T_y, \text{ where } y \text{ is a child of } s_i\}$, $1 \leq i \leq n_s$, and let T' be the subtree of 3-$blk(G)$ obtained from removing the subtree rooted at s_1.

Depending on the degree distribution of vertices in the rooted 3-$blk(G)$, the parallel algorithm treats two separated cases. In case 1, $d_3(s_1) > \frac{l}{4}$. We have a σ-vertex with a high degree. We pick Let W_1 be the first $\min\{d_3(s_1)-2, \lceil\frac{l}{2}\rceil - d_3(s_3)+1\}$ leaves in U_1. Leaves in W_1 are matched with the first $\min\{|W_1|, |U_2|-1\}$ leaves in U_2. Unmatched leaves in W_1, if any, are matched with all remaining leaves but one in T' and finally properly matched with themselves, if necessary. In case 2, $d_3(s_1) \leq \frac{l}{4}$. There is no σ-vertex with a large degree. We show that we can root 3-$blk(G)$ at a vertex u^* with approximately the same number of leaves in each subtree rooted at a child of u^*. If u^* is a β-vertex, a suitable number of leaves between subtrees rooted at children of u^* are matched. If u^* is a σ-vertex, a suitable number of subtrees rooted at children of u^* are first merged into a single subtree rooted at u^*. Then leaves in the merged subtree are matched with leaves outside. Otherwise, u^* is a π-vertex. Let $U = \{u \mid u$ is the leftmost leaf of T_y, where y is a child of $u^*\}$. If $d_3(u^*) > \frac{l}{4}$, we match $\min\{\lceil\frac{l}{2}\rceil - d_3(s_1), \lfloor\frac{d_3(u^*)}{2}\rfloor\}$ pairs of leaves in U. If $d_3(u^*) \leq \frac{l}{4}$, we match a suitable number of leaves between the subtrees rooted at the children of u^*.

The algorithm first finds the matched pairs of leaves in each case. Then we add edges between matched pairs of leaves and update 3-$blk(G)$ at the end of each case. The 3-block tree and the sequence of σ-vertices $s_1, \ldots, s_{n_s}$ will not be changed during the execution of the algorithms for cases 1 and 2.

The parallel algorithm for case 1 and case 2 when u^* is a β-vertex (case 2.1) or a σ-vertex (case 2.2) is similar to the cases for biconnectivity augmentation as described in Chapter 3. In the following subsections, We describe the part for dealing with case 2 when u^* is a π-vertex (case 2.3).

Depending on the degree of u^*, we have two different strategies. We first root 3-$blk(G)$ at u^*. If $d_3(u^*) > \frac{l}{4}$, we find a leaf for each subtree rooted

```
set of pairs of vertices function case2_3_1(tree T);
{* The number of leaves in the 3-block tree T is l. *}
    integer p, k, s; set of pairs of vertices L;
    L := ∅; {* L is the set of matched pairs. *}
    k := min{⌈l/2⌉ − d3(s1), ⌊d3(u*)/2⌋ − 1};
    s := ⌊d3(u*)/2⌋ + 1;
    pfor i = 1 .. k do
        let ui be the ith (clockwise according to the relative position of the ui's
        around the root) child of the root;
        let wi be the leftmost leaf in the subtree rooted at ui;
        let q(wi) be a demanding vertex of wi
    rofp;
    pfor i = 1 .. k do
        L := L ∪ {(q(wi), q(wi+s))}
    rofp;
    return L
end case2_3_1;
```

Algorithm 4.2: Parallel algorithm for handling case 2.3.1.

at a child of u^*. Let the set of leaves found be $\mathcal{U}$. We add an edge between two leaves in $\mathcal{U}$. If $d_3(u^*) \leq \frac{l}{4}$, we add suitable number of edges such that each new edge connects leaves in two subtrees rooted at different children of u^*. We describe the two cases in detail in the following sections.

4.5.1 Case 2.3.1: $d_3(s_1) - 1 \leq \frac{l}{4}$, u^* is a π-vertex, and $d_3(u^*) > \frac{l}{4}$

We first describe the algorithm for this case. Then we prove that each edge we add will decrease the triconnectivity augmentation number by 1 and that we add a constant fraction of the number of edges as indicated by the triconnectivity augmentation number in each execution of Algorithm 4.2.

We find a leaf for each subtree rooted at a child of u^*. Let the set of leaves found be $\mathcal{U}$. We then match $\min\{\lceil \frac{l}{2} \rceil - d_3(s_1), \lfloor \frac{d_3(u^*)}{2} \rfloor - 1\}$ pairs of leaves from $\mathcal{U}$ and add an edge between two matched leaves. See Algorithm 4.2.

Claim 4.5.1 *The number of matched pairs found in function* `case2_3_1` *is at least* $\lfloor \frac{l}{8} \rfloor - 1$.

Proof. Let $k = \min\{\lceil \frac{l}{2} \rceil - d_3(s_1), \lfloor \frac{d_3(u^*)}{2} \rfloor - 1\}$. We add exactly k edges in this case. Since $d_3(u^*) > \frac{l}{4}$, we have $\lfloor \frac{d_3(u^*)}{2} \rfloor - 1 \geq \lfloor \frac{l}{8} \rfloor - 1$. Also, $d_3(s_1) \leq \frac{l}{4}$. Hence $\lceil \frac{l}{2} \rceil - d_3(s_1) \geq \frac{l}{4}$. □

Claim 4.5.2 *Let k be the number of matched pairs found in function* `case2_3_1`. *Let G' be the new graph obtained from G by adding edges between demanding vertices of pairs of leaves found in case 2.3.1. If $l > 11$, then $A_3(G') = A_3(G) - k$ and G' remains balanced.*

Proof. Let k be the total number of edges added and let $p = d_3(u^*)$. Let s be $\lfloor \frac{d_3(u^*)}{2} \rfloor + 1$. Let the set of edges be added in the order (w_1, w_{1+s}), (w_2, w_{2+s}), ..., (w_k, w_{k+s}), and let the graph obtained by adding the first i edges be G_i, $0 \leq i \leq k$. Note that $G_0 = G$. We first prove by induction on i that the pair (w_i, w_{i+s}) satisfies the leaf-connecting condition in 3-$blk(G_i)$.

Induction Base: If $l > 11$, then $d_3(u^*) \geq 4$. Thus the path P between w_1 and w_{1+s} passes through the root, which is a π-vertex with degree at least 4. The path P is non-adjacent on the root. Hence w_1 and w_{1+s} satisfy the leaf-connecting condition in 3-$blk(G_0)$.

Induction Step: Assume the statement is true for any value of i, where $i < r$ and $r > 1$. The structure of 3-$blk(G_{r-1})$ is as follows. Let T_i be the subtree rooted at u_i in 3-$blk(G_0)$, $1 \leq i \leq p$. If we root the updated 3-block tree at the new β-vertex created each time we add an edge, then $T_r, \ldots, T_s$ are in the subtree of 3-$blk(G_{r-1})$ rooted at a π-vertex p_1 and $T_{r+s}, \ldots, T_p$ are in the subtree of 3-$blk(G_{r-1})$ rooted at another π-vertex p_2. The path between p_1 and p_2 passes through the root. Thus the path between w_r and w_{r+s} passes through two vertices of degrees at least 3 (p_1 and p_2).

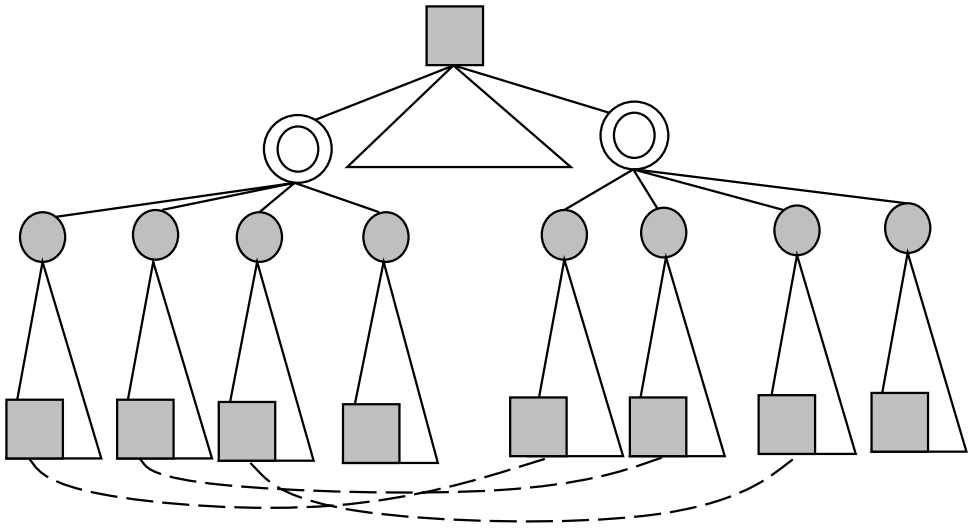


Figure 4.6: This diagram illustrates the proof of correctness for case 2.3.1 for triconnecting a biconnected graph. The 3-block tree obtained after adding the first $r-1$ edges is shown. In the new 3-block tree, only one new β-vertex u is created. Note that the pair of vertices (w_r, w_{r+s}) satisfies the leaf-connecting condition, since the path between them passes through p_1 and p_2, two vertices with degree at least 3.

We have proved that after adding k edges, the number of leaves in the updated 3-block tree decreases by $2k$. Since $k \leq \lceil \frac{l}{2} \rceil - d_3(s_1)$, no σ-vertex in the updated 3-block tree becomes massive. Thus 3-$blk(G)$ remains balanced. (A diagram is given in Figure 4.6 to illustrate this proof.) □

4.5.2 Case 2.3.2: $d_3(s_1) - 1 \leq \frac{l}{4}$, u^* is a π-vertex, and $d_3(u^*) \leq \frac{l}{4}$

Let $v_1, \ldots, v_p$ be the children of u^* such that $x_i \geq x_{i+1}$, $1 \leq i < p$, where x_i is the number of leaves in the subtree rooted at v_i. Let T_i be the subtree rooted at v_i, $1 \leq i \leq p$. We match all but one of the leaves in T_{2i} with the same number of leaves in T_{2i-1}, $1 \leq i \leq \lfloor \frac{p}{2} \rfloor$, using Algorithm 4.3.

Claim 4.5.3 *The number of matched pairs found in function* `case_2_3_2` *is at least* $\frac{l}{8}$.

```
set of pairs of vertices function case_2_3_2 (tree T);
    set of pairs of vertices L;
    let p be the degree of the root of T;
    let v_i, 1 ≤ i ≤ p, be the children of the root u* such that x_i ≥ x_{i+1}, 1 ≤ i < p,
    where x_i is the number of leaves in the subtree rooted at v_i;
    let T_i be the subtree rooted at v_i; L := ∅; {* L is the set of matched pairs. *}
    pfor i = 1 .. ⌊p/2⌋ do
        number leaves in T_{2i-1} from 1 to x_{2i-1} in arbitrary order;
        number leaves in T_{2i} from 1 to x_{2i} in arbitrary order;
        pfor j = 1 .. x_{2i} − 1 do
            let u be the jth leaf of T_{2i-1}; let v be the jth leaf of T_{2i};
            let q(u) and q(v) be a demanding vertex of u and v, respectively;
            L := L ∪ {(q(u), q(v))}
        rofp
    rofp;
    if |L| > ⌈l/2⌉ − d_3(s_1) then remove |L| − (⌈l/2⌉ − d_3(s_1)) pairs from L fi;
    return L
end case_2_3_2;
```

Algorithm 4.3: Parallel algorithm for handling case 2.3.2.

Proof. We know that $x_i \geq x_{i+1}$, $1 \leq i < p$. Thus

$$\sum_{i=1}^{\lfloor \frac{p}{2} \rfloor} x_{2i} \geq \sum_{i=1}^{\lfloor \frac{p}{2} \rfloor} x_{2i+1} \quad \text{and} \quad x_1 + 2\sum_{i=1}^{\lfloor \frac{p}{2} \rfloor} x_{2i} \geq \sum_{i=1}^{p} x_i = l.$$

We have chosen u^* so that no subtree rooted at a child of the root u^* has more than half of the total number of leaves. Thus $x_1 \leq \frac{l}{2}$. Hence

$$\sum_{i=1}^{\lfloor \frac{p}{2} \rfloor} x_{2i} \geq \frac{l}{4}.$$

We know that $p = d_3(u^*) \leq \frac{l}{4}$. Thus

$$\sum_{i=1}^{\lfloor \frac{p}{2} \rfloor} (x_{2i} - 1) \geq \frac{l}{8}.$$

□

Claim 4.5.4 *Let k be the number of matched pairs found in function* case_2_3_2. *Let G' be the new graph obtained from G by adding edges be-*

tween the demanding vertices of pairs of leaves found in case 2.3.2. If $l > 11$*, then* $A_3(G') = A_3(G) - k$ *and* G' *remains balanced.*

Proof. We add an edge between every pair of leaves found in case 2.3.2. Let the set of edges be added in the following order. The set of edges between leaves in T_1 and leaves in T_2 are added first in arbitrary order. Then we add the set of edges between leaves in T_3 and leaves in T_4. We keep on doing this until all edges have been added. Let G be the current graph and let G_i be the graph obtained from G by adding the set of edges between leaves in T_{2i-1} and leaves in T_{2i}. Note that $G_0 = G$. Let $G_{i,j}$ be the graph obtained from G_i by adding j edges between leaves in T_{2i-1} and leaves in T_{2i}. Note that $G_{0,0} = G$. We first prove by induction on i that all pairs matched between leaves in T_{2i-1} and leaves in T_{2i} satisfy the leaf-connecting condition in 3-$blk(G_{i-1})$.

Induction Base: If we add edges to any matched pairs between leaves in T_1 and leaves in T_2, then both T_1 and T_2 have more than two leaves. Thus there is a vertex in 3-$blk(G_0)$ with degree at least 3 on the path from a leaf in T_1 to the root. There is also another vertex with degree at least 3 on the path from a leaf in T_2 to the root. Thus the first pair matched between leaves in T_1 and leaves in T_2 satisfies the leaf-connecting condition. (We show a diagram in Figure 4.7 to illustrate this proof.)

The path $P_{1,i-1}$ between two leaves of the ith pair passes the new β-vertex b_{i-1} created by adding the $(i-1)$th edge. It is either the case that b_{i-1} is degree 4 or b_{i-1} is degree 3 and we can find another degree 3 vertex in $P_{1,i-1}$, since there exists at least one leaf in T_1, and at least one least one leaf in T_2, that is not participating in any matching. Thus each matched pair between a leaf in T_1 and a leaf in T_2 found in case 2.3.2 satisfies the leaf-connecting condition.

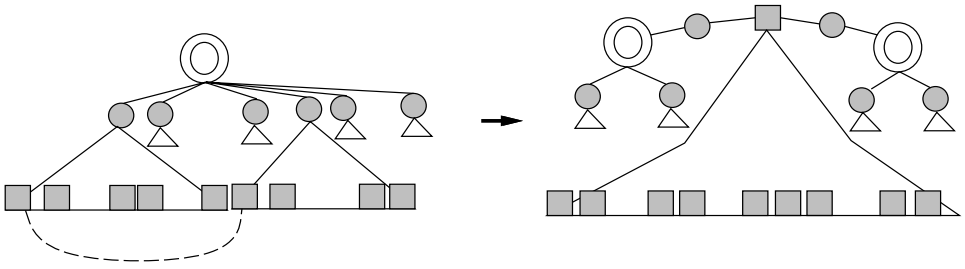


Figure 4.7: This diagram illustrates the proof of correctness for case 2.3.2 for triconnecting a biconnected graph. The 3-block tree for the current graph is shown on the right. The 3-block tree obtained after adding the first edge between the leftmost leaf in T_1 and the leftmost leaf in T_2 is shown on the left. Note that the root of the original 3-block tree is split into two π-vertices p_1 and p_2. Note also that all β-vertices in the path between a_1 and b_1 are collapsed into a new β-vertex u. We root the new 3-block tree at u. Every pair of vertices (a_i, b_i), $1 \leq i < x_2$, satisfies the leaf-connecting condition.

Induction Step: Assume the statement holds for i such that $i < r$ and $r > 1$. The proof for $i = r$ is similar to the base case.

Let r be the total number of edges added. We have proven that after adding k edges, the number of leaves in the updated 3-block tree decreases by $2k$. Since $k < \lceil \frac{l}{2} \rceil - d_3(s_1)$, no σ-vertex in the updated 3-block tree becomes massive. Thus the claim holds. □

4.5.3 The Complete Parallel Algorithm for a Biconnected Graph

We now describe the parallel algorithm (Algorithm 4.4) for the case when the input graph is biconnected. The correctness of this algorithm follows from Claims 4.5.2 and 4.5.4 and Corollaries 3.4.11, 3.4.17 and 3.4.22 given in Chapter 3. We know that the number of leaves in 3-$blk(G)$ is at most n, where n is the number of vertices in the input graph. Algorithm `paug2to3` terminates in $O(\log n)$ iterations and the resulting graph is triconnected.

```
{* G is biconnected with at least 4 vertices;
the algorithm finds a smallest augmentation to triconnect G.  *}
set of pairs of vertices function paug2to3(graph G);
    vertex u*;
    T := 3-blk(G); S := ∅;
    l := number of leaves in T;
    root T at a β-vertex with degree ≥ 2;
    while l > 1 do
      let s1 be a σ-vertex with the largest degree in T;
      if d3(s1) − 1 > l/4 then L := case1(T) {* see Chapter 3 *}
      else {* d3(s1) − 1 ≤ l/4 *}
        root T at an arbitrary vertex;
        find a vertex u* such that there are more than l/2 leaves in the subtree rooted
        at u*, but none of the subtrees rooted at a child of u* has more than l/2 leaves;
        root T at u*;
        permute children of u* (from left to right) in non-increasing order of
        the number of leaves in subtrees rooted at them;
        if u* is a β-vertex then L := case2_1(T) {* see Chapter 3 *}
        else if u* is a σ-vertex then L := case2_2(T) {* see Chapter 3 *}
        else {* u* is a π-vertex *}
          if d3(u*) > l/4 then L := case2_3_1(T)
          else {* d3(u*) ≤ l/4 *} L := case_2_3_2(T) fi
        fi
      fi;
      pfor (a1, a2) ∈ L do
        let ui be a demanding vertex in ai, i ∈ {1, 2};
        S := S ∪ (u1, u2)
      rofp;
1.    update T and l
    od;
    return S
end paug2to3;
```

Algorithm 4.4: Parallel algorithm for finding a smallest triconnectivity augmentation for a biconnected graph.

4.5.4 The Parallel Implementation

In the previous sections, we have shown details of each step in algorithm `paug2to3` except step 1. We now describe an algorithm for updating the current 3-block graph T given the set of edges S added (step 1 in algorithm `paug2to3`).

Updating β-Vertices

If function `paug2to3` executes case 1, case 2.1, or case 2.2, the method for updating β-vertices in the 3-block tree is described in Section 3.5.1. If function `paug2to3` executes case 2.3.1, all β-vertices in a fundamental cycle created by adding edges are merged into one new β-vertex. Let $v_1, \ldots, v_p$ be the children of the root in case 2.3.2. The children are numbered in non-increasing order according to the number of leaves in the subtrees rooted at them. (See Section 4.5.2 for details.) If function `paug2to3` executes case 2.3.2, then at least one of the following two conditions is true: (1) all β-vertices in any fundamental cycle created by adding edges are merged into one new β-vertex; (2) a new β-vertex is created for each set of β-vertices in fundamental cycles that pass through v_{2i-1} and v_{2i}. The following claim gives the method for updating β-vertices and can be easily verified.

Claim 4.5.5 *Let $\{v_1, \ldots, v_p\}$ be the children of the root in 3-blk(G) in case 2.3.2. Let Y_i, $1 \leq i \leq \lfloor \frac{p}{2} \rfloor$, be the set of all vertices in $\{v_1, \ldots, v_p\}$ that are visited if we traverse starting from v_{2i-1} to v_{2i} clockwise around the simple polygon represented by the root of 3-blk(G). We create a new β-vertex (in the new 3-block tree obtained after adding edges found in case 2.3.2) for the set of β-vertices in fundamental cycles passing through v_{2i-1} and v_{2i} if and only if $v_{2j-1} \in Y_i$ implies $v_{2j} \in Y_i$ and vice versa.* □

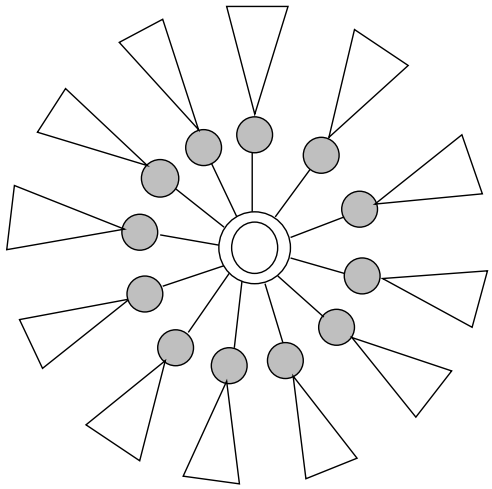


Figure 4.8: This diagram illustrates an example for creating a new β-vertex after adding edges between matched vertices found in case 2.3.2 for triconnecting a biconnected graph. We root the shown 3-block tree at u^*. In case 2.3.2, we match leaves between the subtree rooted at v_{2i-1} and the subtree rooted at v_{2i}. Note that if we traverse from v_{2i-1} to v_{2i} clockwise around the simple polygon represented by the root and we visit v_{2j-1} (v_{2j}), we will also visit v_{2j} (v_{2j-1}). Thus we create one new β-vertex for the set of β-vertices in fundamental cycles that pass through v_{2i-1} and v_{2i}.

Figure 4.8 shows an example where the above condition is satisfied.

Updating σ-Vertices

The method for updating σ-vertices in 3-$blk(G)$ is similar to the method for updating c-vertices in 2-$blk(G)$. For details, see Section 3.5.1.

Updating π-Vertices

The method for updating π-vertices is as follows. For each π-vertex p in 3-$blk(G)$, let

$$S_{1,1}, \ldots, S_{1,r_1}, S_{2,1}, \ldots, S_{2,r_2}, \ldots, S_{2k-1,1}, \ldots, S_{2k-1,r_{2k-1}}, S_{2k,1}, \ldots, S_{2k,r_{2k}}$$

be the σ-vertices that are adjacent to p (in clockwise order). Let

$$\{S_{1,1}, \ldots, S_{1,r_1}, S_{3,1}, \ldots, S_{3,r_3}, \ldots, S_{2k-1,1}, \ldots, S_{2k-1,r_{2k-1}}\}$$

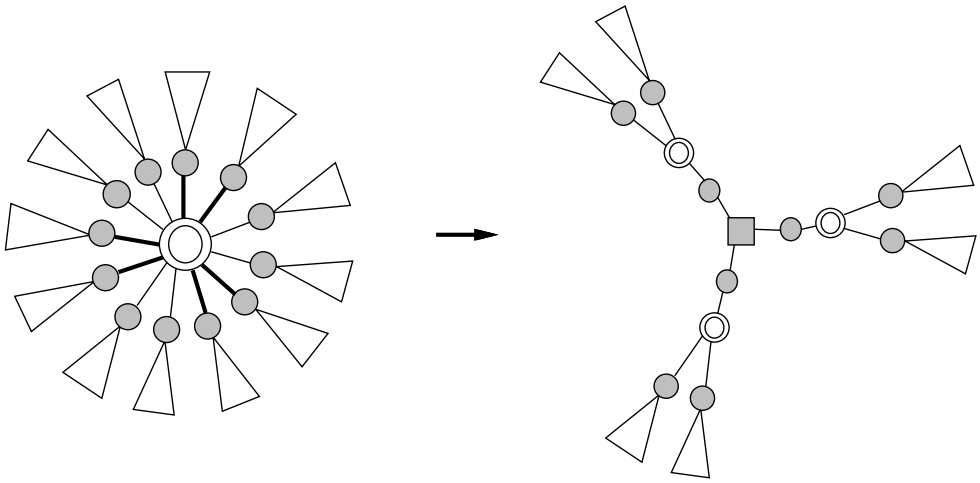


Figure 4.9: Illustrating the procedure for updating a π-vertex p in the 3-block graph after adding a set of edges in parallel. Note that $\{S_{1,1}, S_{1,2}, \ldots, S_{1,r_1}, S_{3,1}, \ldots, S_{3,r_3}, \ldots, S_{2k-1,1}, \ldots, S_{2k-1,r_{2k-1}}\}$ is the set of σ-vertices that are in fundamental cycles created by adding edges. We create a new π-vertex p_i that is adjacent to $S_{2i,j}$, $\forall j$.

be the set of σ-vertices that are in some fundamental cycles created by adding edges. Let $c_{i,j,1}$ and $c_{i,j,2}$ be the two vertices in $S_{i,j}$, for all i and j, such that if we traverse the simple polygon represented by p from $c_{i,j,1}$ clockwise, $c_{i,j,2}$ is visited before any other vertex in $S_{i',j'}$, for all i' and j' such that $\{i', j'\} \neq \{i, j\}$. Notations are shown in Figure 4.9.

The following claim can be easily verified by using properties for updating 3-$blk(G)$.

Claim 4.5.6 *The updated 3-block graph contains a new π-vertex p_i and its adjacent σ-vertices $S_{2i,j}$, $\forall i$, $1 \leq i \leq k$ and $\forall j$, $1 \leq j \leq r_{2i}$. For each new π-vertex p_i, a new σ-vertex S_i is created for the pair of vertices $(c_{2i,1,1}, c_{2i,r_{2i},2})$. S_i is adjacent to p_i and the new β-vertices created.* □

The above updating operations and each iteration of algorithm `paug2to3` can be implemented to run on an EREW PRAM using a linear

number of processors in $O(\log n)$ time by using the parallel merge sort routine in Cole [Col88], the Euler technique in Tarjan & Vishkin [TV85] and procedures in Schieber & Vishkin [SV88]. Algorithm `paug2to3` terminates in $O(\log n)$ iterations. The 3-block tree can be constructed in $O(\log^2 n)$ time using a linear number of processors on an EREW PRAM using routines in [NM82, Ram93]. Thus we have the following claim.

Claim 4.5.7 *Algorithm* `paug2to3` *finds a smallest triconnectivity augmentation for a biconnected graph in* $O(\log^2 n)$ *time on an EREW PRAM using a linear number of processors.* □

4.6 Concluding Remarks

In this chapter, we have presented a linear time sequential algorithm for finding a smallest augmentation to triconnect a biconnected undirected graph. We also have presented an efficient parallel algorithm for this problem. The parallel algorithm runs in $O(\log^2 n)$ time using a linear number of processors on an EREW PRAM. Our parallel algorithm follows the structure of the parallel algorithm for finding a smallest biconnectivity augmentation as described in Chapter 3. However, the 3-block graph data structure used in our algorithm is more complicated than the 2-block graph data structure used in Chapter 3. We had to prove additional properties to derive the desired parallel algorithm. Our parallel algorithm can be made to run within the same time bound using $O(\frac{(n+m)\log\log n}{\log n})$ processors by using the algorithm for finding connected components in [CV86] and the algorithm for integer sorting in [Hag87].

Chapter 5

Smallest Triconnectivity Augmentation: General Graphs

5.1 Introduction

In this chapter, we present a linear time sequential algorithm for finding a smallest augmentation to triconnect any undirected graph. Our sequential algorithm is similar in structure to the one used in biconnectivity augmentation (see Chapter 3). We also show an EREW parallel algorithm for solving this problem in $O(\log^2 n)$ time using a linear number of processors. The approach used for the parallel algorithm is similar to the one in Chapter 3.

Our strategy for finding a smallest triconnectivity augmentation consists of two stages. In stage 1, we biconnect the graph. In stage 2, we use the algorithm in Chapter 4 for optimally triconnecting a biconnected graph. We also make sure that the total number of edges added in these two stages is minimum. It turns out that we cannot use the algorithm for finding a smallest biconnectivity augmentation given in Chapter 3 to implement stage 1, since there exists a graph G such that any smallest augmentation for biconnecting G does not lead to a smallest augmentation for triconnecting G. (See Section 5.3 for details.) Note that for edge-connectivity, it is shown in [NGM90, Wat87] that there exists a smallest augmentation $\mathcal{A}$ to k-edge-connect a graph G such that $\mathcal{A}$ is included in a smallest augmentation to $(k+1)$-edge-connect G, for an arbitrary k. An extended abstract of part of the work reported in this chapter appears in [HR91a].

5.2 Definitions

5.2.1 2-Block Graphs

We give a modified definition for the 2-block graph here that is not exactly the same as the one given in Chapter 3 and in [ET76, RG77]. In Chapter 3 and [ET76, RG77], edges are partitioned into maximal sets such that any two distinct edges in the same set are in a simple cycle. Each partition of edges is called a *2-block*. Thus a cut edge is represented by a *b*-vertex in the block graph given in Chapter 3 and in [ET76, RG77]. The intersection of any two 2-blocks is a cutpoint. The above definition was changed to guarantee that a non-trivial 2-block is biconnected. Our revised definition is based on vertex-partitions instead of edge-partitions. The two definitions are very similar. We can easily transform one to the other. Operations and properties defined in one can also be easily modified to apply on the other.

2-Block [ET76, Eve79, RG77]

Given an undirected graph G with a set of vertices V, let $\mathcal{V} = \{V_i \mid 1 \leq i \leq k\}$ be a set of subsets of V such that $\cup_{i=1}^{k} V_i = V$ and two vertices u and w are in the same subset if and only if there is a simple cycle in G which contains u and w or $u = w$. The induced subgraph of G on each V_i, $1 \leq i \leq k$, is a *2-block*. The union of all 2-blocks includes all edges in G that are not *cut edges* (an edge is called a *cut edge* in G if its removal makes the resulting graph contain more connected components than those in G). For a graph G with h connected components, a vertex w is a *cutpoint* of G if and only if the graph obtained from G by removing w and all edges adjacent to w contains more than h connected components. A 2-block containing only one vertex is called a *trivial 2-block*. A trivial *2-block* of G is called a *cut-block* if it is a cutpoint. It is easy to see that a trivial 2-block is not a cut-block if and only if it contains a vertex of degree

less than 2 in G. It is well-known that a 2-block is biconnected if it contains at least three vertices.

2-Block Graph

Given an undirected graph G, we define its *2-block graph*, 2-$blk(G)$, as follows. Each cutpoint and 2-block that is not a cut-block is represented by a vertex in 2-$blk(G)$. The vertices of 2-$blk(G)$ that represent blocks that are not cutpoints are called *b-vertices* and those representing cutpoints are called *c-vertices*. For a vertex u in 2-$blk(G)$, let $V_u = \{u\}$ if u is a c-vertex and $V_u = \{w \mid w$ is a vertex in the corresponding 2-block represented by $u\}$ if u is a b-vertex. The induced subgraph of G on V_u is the *corresponding subgraph* of u. Two vertices u and w in 2-$blk(G)$ are adjacent if and only if any one of the following conditions is true: (i) $|V_u| = 1$, $|V_w| = 1$ and the vertex in V_u and the vertex in V_w are adjacent in G; (ii) $|V_u| = 1$ and $V_u \subset V_w$; (iii) $|V_w| = 1$ and $V_w \subset V_u$. It is well-known that 2-$blk(G)$ is a forest. A degree-1 b-vertex in 2-$blk(G)$ is a *2-block leaf*. If G is connected, 2-$blk(G)$ is a tree. If 2-$blk(G)$ is a tree, we refer to it as a *2-block tree*. For a vertex v in G, let $d_2(v)$ be the degree of its corresponding c-vertex in 2-$blk(G)$ if v is a cutpoint. If v is not a cutpoint, let $d_2(v) = 1$. For more on the properties of 2-$blk(G)$, see Chapter 3. An example of a graph and its 2-$blk(G)$ is shown in Figure 5.1, where rectangles and circles represent b-vertices and c-vertices, respectively.

5.2.2 3-Block Graphs

We give an extended definition for the 3-block graph (which was given in Chapter 4 for a biconnected input graph) to handle the case when the input graph is not biconnected.

3-Block Graph

Given a 2-block H of G, let 3-$blk(H)$ be its 3-block graph. If H is a trivial

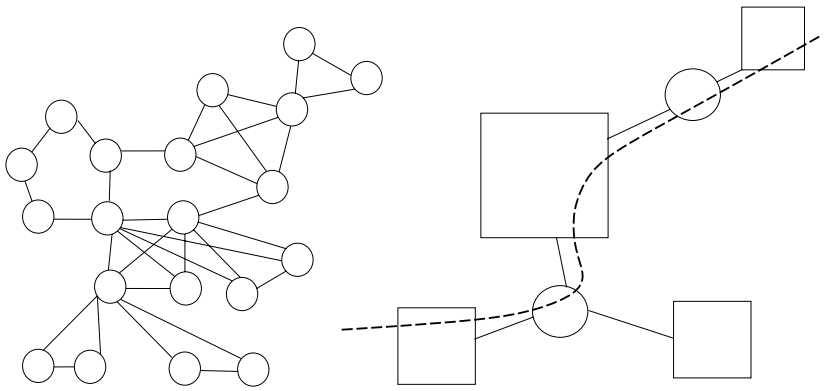


Figure 5.1: A graph G, 2-$blk(G)$, and the implied path between vertices 16 and 21 in the 2-block graph (to be defined later). We represented a b-vertex and a c-vertex, by a rectangle and a circle, respectively. The vertex-number appearing within each c-vertex in 2-$blk(G)$ represents the corresponding cutpoint in G. The vertex-numbers appearing with each b-vertex in 2-$blk(G)$ represent the vertices in its corresponding 2-block.

2-block, let 3-$blk(H)$ be a single β-vertex corresponding to the *trivial Tutte component* H.

From [HT73, Ram93, Tut66], we know that 3-$blk(H)$ is a tree. We call this tree a *3-block tree*. We call the set of trees corresponding to 2-blocks in G *the 3-block graph* of G or 3-$blk(G)$. Each Tutte component that corresponds to a β-vertex in the 3-block graph is a *3-block* of G. We know that 3-$blk(G)$ is a tree if G is biconnected. Given a graph G with n vertices and m edges, 2-$blk(G)$ and 3-$blk(G)$ can be computed in $O(n+m)$ time using procedures in [HT73, Ram93]. An example of a 3-block graph is shown in Figure 5.2. (We use the notation given in Chapter 4 to illustrate 3-block graphs.)

The Implied Path in the 2-Block Graph

Let G be an undirected graph and let u and v be two distinct vertices in G. Let G' be the graph obtained from G by adding the edge (u, v) and let b_u and b_v be the two vertices in 2-$blk(G)$ whose corresponding subgraphs contain u

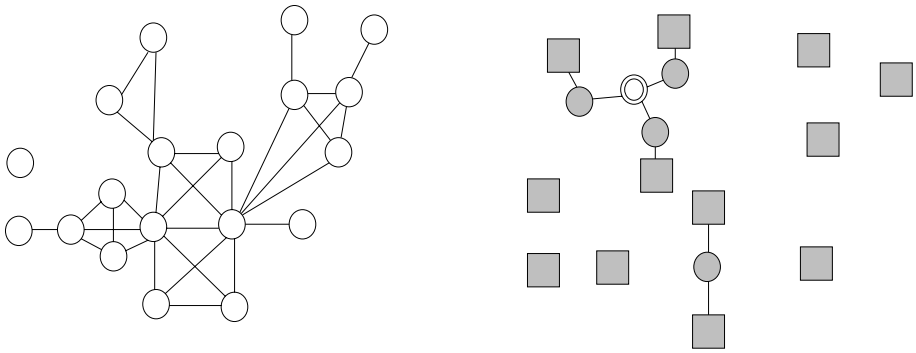


Figure 5.2: A graph G and its 3-block graph. The β-vertices corresponding to the sets of vertices {4, 10, 11, 12} and {6} are 3-block leaves (to be defined later), while the β-vertices corresponding to the sets of vertices {4, 8, 9, 10} and {8} are not. The β-vertices corresponding to the sets of vertices {17}, {19} and {2, 3, 4, 5} are isolated 3-block vertices (to be defined later), while the β-vertex corresponding to the set of vertices {10, 14, 15, 16} is not.

and v, respectively. We denote the path P_2 between b_u and b_v in 2-$blk(G)$ *the implied path between u and v in 2-$blk(G)$* (we let $P_2 = [b_u]$ if there is no such path). An example is shown in Figure 5.1.

Separating-Sequence, Cut-Sequence and the Collection of Implied Paths in the 3-Block Graph

Let Y be the set of all b-vertices in the implied path P_2 and those c-vertices in P_2 whose corresponding cutpoints are cut-blocks. Let $|Y| = r$ and y_i be the ith vertex in Y encountered when we traverse P_2 starting form b_u to b_v. We construct a *separating-sequence* $W = [\, w_1 = u, w_2, \ldots, w_{2r-1}, w_{2r} = v]$ for vertices u and v such that for all i, $1 < i < r$, w_{2i-1} and w_{2i} are the two cutpoints adjacent to y_i if y_i is a b-vertex; w_{2i-1} and w_{2i} are the two vertices adjacent to y_i in G if y_i is a cutpoint; w_2 and w_{2r-1} are cutpoints adjacent to y_1 and y_r, respectively. Note that w_{2i} and w_{2i+1} can be the same, $1 \leq i < r$; they are different if and only if any there is at least one c-vertex in $\{y_i, y_{i+1}\}$. A *cut-sequence* $C = [u, c_1, \ldots, c_x, v]$ for vertices u and v is a sequence of vertices

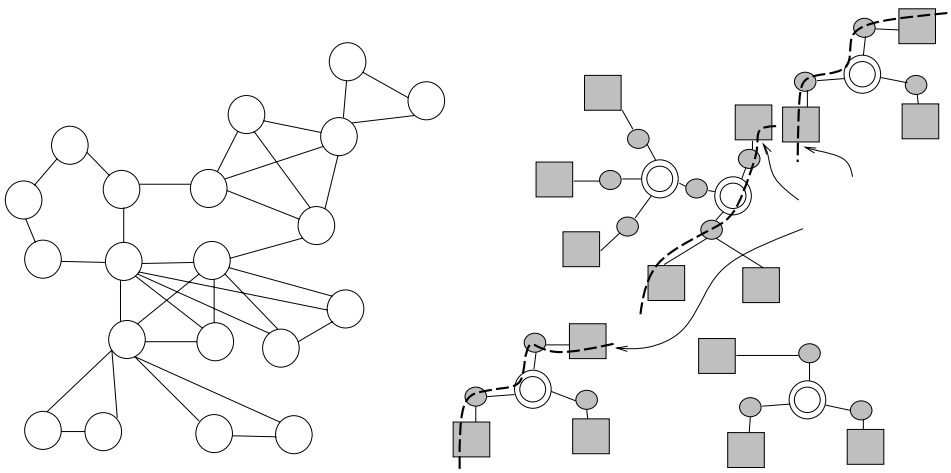


Figure 5.3: A graph G and 3-$blk(G)$ (including 4 trees). The *separating-sequence* between vertices 16 and 21 is [16, 6, 6, 12, 12, 21]. The *cut-sequence* between vertices 16 and 21 is [16, 6, 12, 21]. The collection of implied paths between vertices 16 and 21 in 3-$blk(G)$ is also illustrated.

in G such that c_i is the cutpoint corresponding to the ith c-vertex encountered when we traverse P_2 from b_u to b_v and x is the number of c-vertices in P_2. An example is shown in Figure 5.3.

For $1 \leq i \leq r$, let H_i be the corresponding 2-block represented by y_i if y_i is b-vertex; let H_i be the corresponding cutpoint represented by y_i if y_i is a c-vertex. For $1 \leq i \leq r$, let $Q_i = [H_i]$ if y_i is a c-vertex; otherwise, let Q_i be the path in 3-$blk(H_i)$ between the two β-vertices that correspond to Tutte components containing w_{2i-1} and w_{2i}. Each Q_i, $1 \leq i \leq r$, is chosen such that both w_{2i-1} and w_{2i} are each contained in exactly one of the Tutte components corresponding to β-vertices in Q_i if y_i is a b-vertex. Note that w_{2i} and w_{2i+1} can be the same, $1 \leq i < r$; they are different if and only if there are a sequence of more than one cutpoint between y_i and y_{i+1} in P_2. If there is more than one Tutte component that contains w_i, $1 \leq i \leq 2r$, we choose an arbitrary one. We call $\mathcal{Q} = \{Q_1, \ldots, Q_r\}$ *the collection of implied paths between u and v in*

3-blk(G). If G is biconnected, $r = 1$; Q_1 is also called *the implied path between* u *and* v *in 3-blk*(G). An example of a collection of implied paths in 3-*blk*(G) is shown in Figure 5.3.

Separating Degrees for σ-Vertices

Given a graph with h connected components, the *separating degree* $sd((a_1, a_2))$ of a Tutte pair $(a_1,\ a_2)$ is $\sum_{i=1}^{2} d_2(a_i) + d_3((a_1, a_2)) + h - 4$. We will show (Claim 5.3.1 in Section 5.3) that the separating degree of a Tutte pair is equal to the smallest number of edges needed to connect the graph obtained from G by removing the Tutte pair. The separating degree for the corresponding σ-vertex s, $sd(s)$, is equal to $sd((a_1, a_2))$.

5.3 A Lower Bound for the Triconnectivity Augmentation Number

We state a claim to justify the way the separating degree of a Tutte pair in a 3-block graph is defined.

Claim 5.3.1 *The separating degree of a Tutte pair is equal to the smallest number of edges needed to connect the graph obtained from* G *by removing the Tutte pair.*

Proof. Let G be a graph with h connected components. Recall that the separating degree $sd(s)$ for a Tutte pair $(a_1,\ a_2)$ in 3-*blk*(G) is equal to $\sum_{i=1}^{2} d_2(a_i)$ $+ d_3(s) + h - 4$. Let G_1 be the connected component in G that contains a_1 and a_2. Let Q be the set of connected components in the graph obtained from G_1 by removing a_1 and a_2. We can partition Q into three disjoint sets Q', Q_1 and Q_2 such that $Q' = \{C \mid C \in Q, C$ contains a vertex adjacent to a_1 and a vertex adjacent to $a_2\}$; $Q_1 = \{C \mid C \in Q, C$ contains a vertex adjacent to $a_1\} - Q'$; Q_2

$= \{C \mid C \in Q, C$ contains a vertex adjacent to $a_2\} - Q'$. From the definition of 2-block graph and 3-block graph, we know that $|Q'| = d_3(s)$, $|Q_1| = d_2(a_1) - 1$ and $|Q_2| = d_2(a_2) - 1$. Thus the graph G' obtained from G by removing a_1 and a_2 contains $\sum_{i=1}^{2}(d_2(a_i) - 1) + d_3(s) + h - 1$ connected components. Hence it takes at least $\sum_{i=1}^{2} d_2(a_i) + d_3(s) + h - 4$ edges to connect G'. □

We identify β-vertices in a 3-block graph whose corresponding Tutte components must contain a new incoming edge if we want to triconnect the input graph.

Definition 5.3.2 *Given a β-vertex b in 3-blk(G), let H_b be its corresponding Tutte component of G. A degree-1 β-vertex b in 3-blk(G) is a* **3-block leaf** *if any one of the following conditions is true:*
(i) H_b consists of only one vertex u and u is not a cutpoint; (Note that (i) holds if and only if u is in a polygon.)
(ii) if H_b contains any cutpoint c of G, c is in a Tutte pair of G contained in H_b.

A degree-0 β-vertex b in 3-blk(G) is an **isolated 3-block vertex** *if any one of the following conditions is true:*
(i) H_b contains at most 2 cutpoints;
(ii) H_b consists of only one vertex u and the degree of u in G is at most 2.

Let H be a nontrivial Tutte component that corresponds to a β-vertex in 3-*blk*(G). From Definition 5.3.2 we know that G becomes disconnected if we remove the following vertices from G: (i) vertices in H that are cutpoints of G; (ii) vertices in H that are in a Tutte pair in H. If H is a trivial Tutte component, the graph obtained from G by removing vertices that are adjacent to H is disconnected. Let r be the number of vertices removed. We know that

$r \leq 2$ if H corresponds to a 3-block leaf or an isolated 3-block vertex. If we want to triconnect the input graph, we must add at least $3 - r$ new edges incoming to H. Figure 5.2 illustrates 3-block leaves and isolated 3-block vertices. Note that if G is biconnected, then a degree-1 β-vertex in 3-$blk(G)$ must be a 3-block leaf; this is not necessarily true if G is not biconnected.

We now identify demanding vertices in H that are analogous to demanding vertices defined in Chapter 4.

Definition 5.3.3 *Given a 3-block leaf or an isolated 3-block vertex b in 3-$blk(G)$, let H_b be its corresponding Tutte component. A vertex u in H_b is a* **demanding vertex** *of b if any one of the following conditions is true: (i) u is not a cutpoint or in any Tutte pair of G contained in H_b; (ii) b is an isolated 3-block vertex and H_b consists of only the vertex u. The vertex u is also called a demanding vertex in G.*

Claim 5.3.4 *There exists at least one demanding vertex for any 3-block leaf or isolated 3-block vertex in a 3-block graph.*

Proof. The corresponding Tutte component H_b for b that is a 3-block leaf can be a vertex with degree 2 in G or a triconnected component with a single Tutte pair. If H_b is a single vertex, then the claim is true. If H_b is triconnected, H_b consists of at least 4 vertices. Since H_b contains only one Tutte pair, the claim holds.

If b is an isolated 3-block vertex with H_b being a triconnected component, then the claim can be proved in a way similar to the one given above. If H_b contains less than 4 vertices, then H_b must contain a single vertex y. From Definition 5.3.3, we know that y is the demanding vertex for b. □

When we specify the 3-block graph of a graph G, we will include a demanding vertex to each 3-block leaf and each isolated 3-block vertex in the 3-block graph.

We now define the *weight* of a graph, which we will relate later to the number of edges needed to triconnect the graph.

Definition 5.3.5 *Let H be a 2-block in a graph G whose corresponding vertex in 2-blk(G) is r_H. Let $l_3(H)$ be the number of 3-block leaves in H. We define the* **weight** *of the graph G, $w(G)$, to be $\sum_{\forall\ 2\text{-blocks } H} \max\{3-d_2(r_H), l_3(H)\}$, if G is not biconnected. Otherwise, let $w(G) = l_3(G)$.*

Note that in Chapter 4, the weight of a biconnected graph G is defined as $l_3(G)$, which is equivalent to the above definition.

We now state a lower bound for the triconnectivity augmentation number for a general undirected graph.

Lemma 5.3.6 *Given an undirected graph G, we need at least $\max\{d, \lceil \frac{w(G)}{2} \rceil\}$ edges to triconnect G, where d is the largest separating degree among all σ-vertices in 3-blk(G).*

Proof. The first component of the lower bound comes from Claim 5.3.1. For the other component of the lower bound, suppose that G' is triconnected and is obtained from G by adding a smallest set of edges. For each 2-block H, we must have at least $3-d_2(r_H)$ new incoming edges in G'. For each 3-block leaf in 3-$blk(H)$, the induced subgraph on vertices corresponding to it must contain at least one new incoming edge in G'. Hence we need at least $\lceil \frac{\max\{3-d_2(r_H), l_3(H)\}}{2} \rceil$ new edges for each 2-block H. The total number of new edges needed is thus

$\lceil \frac{w(G)}{2} \rceil$. Hence we need at least $\max\{d, \lceil \frac{w(G)}{2} \rceil\}$ edges in G'. This proves the lemma. □

Note that the lower bound stated in Lemma 5.3.6 is equal to $\max\{d, \lceil \frac{l_3(G)}{2} \rceil\}$ when the graph is biconnected. For biconnected graphs, this is equivalent to the lower bound given in Chapter 4. Note also that it is shown in [NGM90, Wat87] that there exists a smallest augmentation $\mathcal{A}$ to k-edge-connect a graph G such that $\mathcal{A}$ is included in a smallest augmentation to $(k+1)$-edge-connect G, for an arbitrary k. Note that a linear time sequential algorithm for finding a smallest augmentation to triconnect a biconnected graph is presented in Chapter 4. We would like to know whether we can design an algorithm for finding a smallest augmentation to triconnect a general graph by first finding a smallest augmentation to biconnect it (a linear time algorithm is described in Chapter 3) and then applying the algorithm to triconnect it. The following lemma states that this is not always possible.

Lemma 5.3.7 *There exists a graph G such that any smallest augmentation for biconnecting G does not lead to a smallest augmentation for triconnecting G.*

Proof. Consider the graph G shown in Figure 5.4. From Chapter 3 and [ET76], we know that the smallest number of edges needed to biconnect G is 2. To biconnect G, we must first add an edge between a vertex in $\{1,2\} \cup \{3,4,5,6,7,8,9\}$ and a vertex in $\{12,13\} \cup \{15,16\}$. Without loss of generality, we assume the edge added is between a vertex in {1, 2} and a vertex in {12, 13}. The next edge added must be between a vertex in {3, 4, 5, 6, 7, 8, 9} and a vertex in {15, 16}. Since the Tutte pair {9, 10} has a separating degree 6 in the resulting biconnected graph G', we need to add at least 5 more edges to

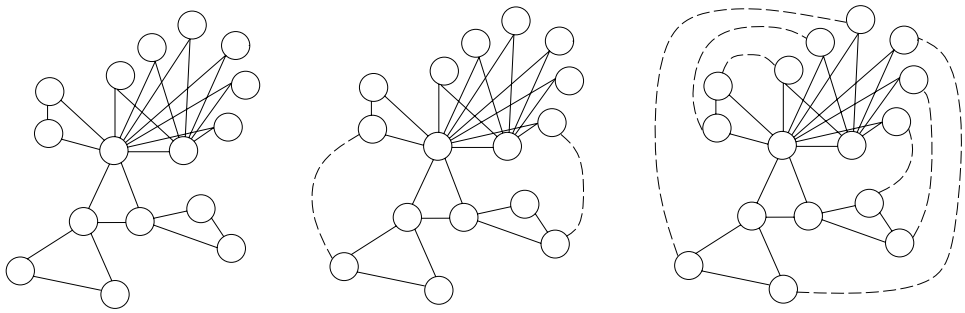


Figure 5.4: An example that shows that finding any smallest augmentation to biconnect a graph G (the graph on the left) does not lead to a smallest augmentation for triconnecting G.

triconnect G'. (Lemma 5.3.6). The total number of edges used is 7. However, it takes only 6 edges to triconnect the original graph as shown in Figure 5.4. This proves the lemma. □

5.4 Finding a Smallest Augmentation to Triconnect a Graph

In this section, we consider the problem of finding a smallest augmentation to triconnect any undirected graph. We show that the lower bound given in Lemma 5.3.6 can be always achieved by giving an algorithm for finding a smallest triconnectivity augmentation with the required cardinality. Finally, we describe a linear time implementation for the algorithm.

5.4.1 Properties of the 3-Block Graph

In this section, we study properties of the 3-block graph that are used in the next section for designing an algorithm to find a smallest triconnectivity augmentation.

Massive, Critical and Balanced

For an undirected graph G with weight $w(G)$, a Tutte pair z or its corresponding

σ-vertex is *massive* if $sd(z) > \lceil \frac{w(G)}{2} \rceil$. A Tutte pair z or its corresponding σ-vertex is *critical* if $sd(z) = \lceil \frac{w(G)}{2} \rceil$. If no Tutte pair in G is *massive*, then G and its 3-block graph are called *balanced*.

In Lemma 5.4.1, we study the changes made on the 3-block graph after adding an edge into the original graph. We then state the property of the set of all critical σ-vertices in 3-$blk(G)$ using Claim 5.4.5; the condition that after adding an edge, the separating degree of certain σ-vertices decreases by one in Claim 5.4.7; the condition that after adding an edge, the weight of the resulting graph decreases by two in Lemma 5.4.8. Finally in Corollary 5.4.9, we show that we can always decrease the lower bound given in Lemma 5.3.6 by one by adding an edge.

Let G' be the graph obtained from G by adding an edge between two demanding vertices u and v in G. We describe the updating operation performed on 3-$blk(G)$ to get 3-$blk(G')$ after adding an edge.

Lemma 5.4.1 *Let G' be the graph obtained from G by adding an edge between two demanding vertices u and v in G, where u and v are in different Tutte components. Let b_u and b_v be the two b-vertices in 2-$blk(G)$ whose corresponding 2-blocks contain u and v, respectively. Let Y be the set of b-vertices and c-vertices whose corresponding cutpoints are cut-blocks in P_2, the path in 2-$blk(G)$ between b_u and b_v. Let $|Y| = r$ and y_i be the ith vertex in Y encountered when we traverse P_2 starting form b_u to b_v. Given $\{Q_1, \ldots, Q_r\}$, the collection of implied paths in 3-$blk(G)$ between u and v, the separating-sequence $[w_1, \ldots, w_{2r}]$ and the cut-sequence [u, $c_1, \ldots, c_x$, v], we can obtain 3-$blk(G')$ from 3-$blk(G)$ by performing the following operations.*

(i) For each Q_i, $1 \leq i \leq r$, we perform the operation given in Chapter 4 on the 3-block tree that contains Q_i if Q_i contains more than 1 vertex.

(ii) A new π-vertex is created with the corresponding Tutte component being the simple cycle [u, $c_1, \ldots, c_x$, v, u]. Each pair of vertices w_{2i-1} and w_{2i}, $1 \leq i \leq r$, is a Tutte pair. The corresponding σ-vertex for Tutte pair (w_{2i-1}, w_{2i}) is incident on the new π-vertex created.

(iii) For each c-vertex y_i, $1 \leq i \leq r$, we create a β-vertex and connect it to the σ-vertex corresponds to Tutte pair (w_{2i-1}, w_{2i}).

(iv) For each b-vertex y_i, $1 \leq i \leq r$, the σ-vertex corresponding to the Tutte pair (w_{2i-1}, w_{2i}) is incident on the β-vertex z whose corresponding Tutte component contains w_{2i-1} and w_{2i} if z exists.

(v) We merge σ-vertices corresponds to the same Tutte pair and delete degree-1 π-vertices.

Proof. The correctness of performing operations in part (i) is given in Chapter 4. Part (ii) is from the definition of Tutte split and the definition of the 3-block graph. Parts (iii), (vi) and (v) are from the definition of the 3-block graph. □

An example is shown in Figure 5.5.

We know that if G is biconnected, there can be at most two critical σ-vertices in 3-$blk(G)$ as described in Chapter 4. But this is not true for G is not biconnected (see also [WN88]). We now state a claim about the set of all critical σ-vertices in 3-$blk(G)$ for a graph G. In general, if there are more than two critical σ-vertices in 3-$blk(G)$, we can partition the set of critical σ-vertices into at most two subsets such that either a subset has exactly one σ-vertex or Tutte pairs corresponding to critical σ-vertices in a subset share a common cutpoint.

In the statement of the following lemmas and claim, we use the following notations. Let G be a connected graph. Let $\mathcal{S} = \{s_1, \ldots, s_r\}$ be the set

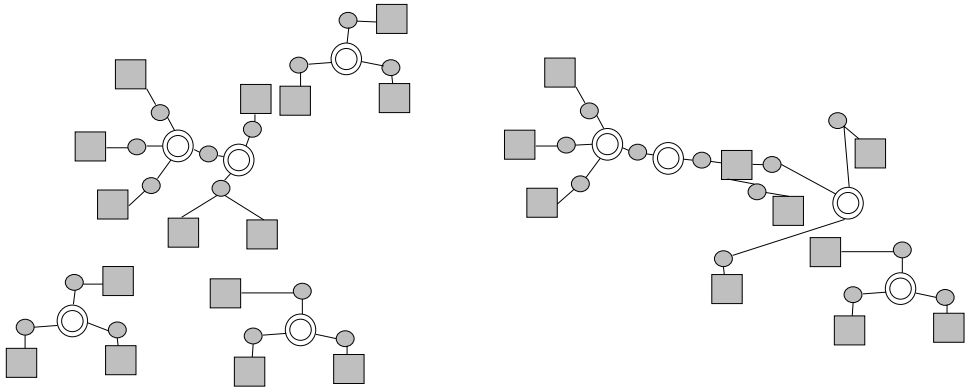


Figure 5.5: A 3-*blk*(G) (including 4 trees) whose original graph G is shown in Figure 3, and the updated 3-block graph after adding a new edge between vertices 16 and 21. The *separating-sequence* between vertices 16 and 21 is [16, 6, 6, 12, 12, 21]. The *cut-sequence* between vertices 16 and 21 is [16, 6, 12, 21]. The new π-vertex q is created for the cut sequence, which represents the polygon [16, 6, 12, 21]. The new 3-block graph contains only 2 trees.

of critical σ-vertices in 3-*blk*(G). Given $\mathcal{S}$, let $\Im_1 = \{c \mid c$ is a cutpoint that is shared by more than one Tutte pair represented by members of $\mathcal{S}\}$. Let $\Im_2 = \{s \mid s \in \mathcal{S}$ and $\nexists c \in \Im_1$ such that the Tutte pair represented by s contains $c\}$. The proofs of Lemma 5.4.2, Lemma 5.4.3 and Lemma 5.4.4 are quite lengthy. We prove them by discussing the lower bound of the weight of G for different values of $|\Im_1|$ and $|\Im_2|$. The details of the proofs can be found in Appendix A.

Lemma 5.4.2 $|\Im_1| \leq 2$. *If* $|\Im_1| = 2$, *then each Tutte pair in* G *contains one of the cutpoints in* $\Im_1$ *and there are only two cutpoints in* G.

Proof. See Appendix A. □

Lemma 5.4.3 *If* $|\Im_1| = 1$, *then* $|\Im_2| \leq 1$. *If* $\Im_1 = \{c\}$ *and* $|\Im_2| = 1$, *where* c *is a cutpoint, then all vertices in 2-*$blk(G)$ *have degrees less than or equal to 2.*

Proof. See Appendix A. □

Lemma 5.4.4 *Let G be a connected graph with $w(G) > 2$. Then $|\Im_2| \leq 2$. If $\Im_2 = \{s_1, s_2\}$, where s_1 and s_2 are critical Tutte pairs, then*
(i) G is biconnected if s_1 and s_2 are in the same 2-block;
(ii) there are only two cutpoints in G if s_1 and s_2 are in different 2-blocks. The two cutpoints are connected by a cut edge.

Proof. See Appendix A. □

The following claim follows directly from Lemma 5.4.2, Lemma 5.4.3 and Lemma 5.4.4.

Claim 5.4.5 $|\Im_1| + |\Im_2| \leq 2$. □

The following is a corollary of Claim 5.4.5.

Corollary 5.4.6 *Let G be an undirected graph. We can find two demanding vertices u_1 and u_2 in G such that each critical σ-vertex is either in the collection of implied paths between u_1 and u_2 in 3-blk(G) or its corresponding Tutte pair contains a cutpoint in the implied path between u_1 and u_2 in 2-blk(G).* □

The following claim states some conditions under which separating degrees of certain σ-vertices decrease by 1.

Claim 5.4.7 *Let G be an undirected graph and let u and v be two demanding vertices in G. Let G' be the graph obtained from G by adding the edge (u, v) and let s be a σ-vertex in 3-blk(G) with $sd(s) \geq 3$. Then $sd(s)$ decreases by 1 in G' if any one of the following conditions is true:*
(i) s is in one of the paths in the collection of implied paths between u and v in 3-blk(G);
(ii) the corresponding Tutte pair of s contains a cutpoint in the implied path between u and v in 2-blk(G).

Proof. Let s be a σ-vertex with $sd(s) \geq 3$ and let s lie in one of the paths in the collection of implied paths between u and v in 3-$blk(G)$ (part (i)). From Chapter 4, we know two edges adjacent to s are eliminated and one new edge is created and connected to s. Thus the degree of s decreases by one.

Let s be a σ-vertex with $sd(s) \geq 3$ and let its corresponding Tutte pair be (a_1, a_2), where a_1 is in the implied path P_2 in 2-$blk(G)$ (part (ii)). Thus a_1 is a cutpoint. If P_2 also passes through a_2, then a_2 is also a cutpoint; from the properties of the 2-block graph proved in Chapter 3, we know that $d_2(a_1)$ and $d_2(a_2)$ both decrease by one. Since $d_3(s)$ increases by one in G', $sd(s)$, which is equal to $\sum_{i=1}^{2} d_2(a_i) + d_3(s) + h - 4$, decreases by 1. If P_2 does not pass through a_2, then $d_2(a_1)$ decreases by 1 and $d_2(a_2)$ and $d_3(s)$ both remain the same. Thus $sd(s)$ is decreased by one. This proves the claim. □

Lemma 5.4.8 states a condition under which the weight of the graph decreases by 2 after adding an edge.

Lemma 5.4.8 *Let G' be the graph obtained from a graph G by adding an edge between two distinct demanding vertices u and v. Then $w(G') = w(G) - 2$ if u and v are in different connected components or in different 2-blocks.*

Proof. Let G be a graph with connected components $G_1, \ldots, G_h$ and let u and v be in G_q and G_r, respectively. Let G'_q be the graph obtained from $G_q \cup G_r$ by adding (u, v). Let u and v be in 2-blocks H_q and H_r, respectively and let H'_q and H'_r be 2-blocks of G' that contain u and v, respectively. Let r_q and r_r be b-vertices in 2-$blk(G)$ that represent H_q and H_r, respectively and let r'_q and r'_r be b-vertices in 2-$blk(G')$ that represent H'_q and H'_r, respectively.

Since u and v are in different connected components or in different 2-blocks, G is not biconnected. If G' is biconnected, G is connected and u

and v are in different 2-blocks. Thus for each $i \in \{q, r\}$, $l_3(H_i') = l_3(H_i) - 1$ if $l_3(H_i) > 0$; $l_3(H_i') = 1$ and $3 - d_2(r_i) = 2$ if $3 - d_2(r_i) > l_3(H_i)$. Since $w(G') = l_3(H_q')$ and $w(G) = \sum_{i \in \{q,r\}} \max\{3 - d_2(r_i), l_3(H_i)\}$ if G' is biconnected, $w(G') = w(G) - 2$. The case for G_q' is biconnected can be proved in a similar way.

We now prove this lemma for the case when G_q' is not biconnected (that is, G and G' are not biconnected). We know that $w(G) = \sum_{i=1}^{h} w(G_i)$ and $w(G') = w(G) - w(G_q) - w(G_r) + w(G_q')$ if $q \neq r$; $w(G') = w(G) - w(G_q) + w(G_q')$ if $q = r$. For each $i \in \{q, r\}$, $l_3(H_i') = l_3(H_i) - 1$ if $l_3(H_i) > 0$; $3 - d_2(r_i') = (3 - d_2(r_i)) - 1$ if $3 - d_2(r_i) > l_3(H_i)$. Thus $\max\{3 - d_2(r_i'), l_3(H_i)\}$ $= \max\{3 - d_2(r_i'), l_3(H_i)\} - 1$, $i \in \{q, r\}$. Hence $w(G') = w(G) - 2$. This proves the lemma. □

By Lemma 5.4.8 and Corollary 5.4.6, we now show that for any biconnected graph, we can always decrease the lower bound given in Lemma 5.3.6 by 1 by adding an edge.

Corollary 5.4.9 *Let G be a connected graph that is not biconnected. We can find two demanding vertices u_1 and u_2 in G such that they satisfy the condition given in Corollary 5.4.6 and $w(G \cup \{(u_1, u_2)\}) = w(G) - 2$.*

Proof. If we can find two demanding vertices not in the same 2-block that satisfy the condition given in Corollary 5.4.6 or any two critical σ-vertices whose Tutte pairs share a common cutpoint, then Lemma 5.4.8 shows $w(G \cup \{(u_1, u_2)\}) = w(G) - 2$. Let u_1 and u_2 be in the 2-block H; every critical σ-vertex is also in H and they do not share a cutpoint. From Claim 5.4.5, we know that there can be at most two critical σ-vertices whose Tutte pairs do not share a cutpoint. If there is only one critical σ-vertex in H, we can easily

substitute one of u_1 and u_2 for a demanding vertex in a 2-block that is not H such that u_1 and u_2 satisfy both the conditions given in Corollary 5.4.6 and Lemma 5.4.8. Thus $w(G \cup \{(u_1, u_2)\}) = w(G) - 2$. If there are two critical σ-vertices in H, their degrees in 3-$blk(G)$ must be greater than 2 since G is not biconnected and they are the only two critical σ-vertices in 3-$blk(G)$. The number of 3-block leaves decreases by 2 as described in Chapter 3. Thus the weight of G decreases by 2 after adding (u_1, u_2). □

5.4.2 An Algorithm for Triconnecting an Undirected Graph Using the Smallest Number of Edges

In the following algorithm, `aug3`, we find a pair of demanding vertices u and v in G such that the collection of implied paths $\mathcal{Q}$ in 3-$blk(G)$ between u and v passes through the massive σ-vertex or all critical σ-vertices if G is connected. If G is not connected, u and v are demanding vertices in G that are in different connected components. We also have to make sure that $w(G)$ is reduced by 2 if 3-$blk(G)$ is balanced by satisfying the conditions given in Lemma 5.4.8. Before the proof of correctness for algorithm `aug3`, we state a claim for an input graph G that is unbalanced. The proof of this claim is similar to a proof in Chapter 3 and [RG77].

Claim 5.4.10 *If there exists a massive σ-vertex s_1 in 3-$blk(G)$, then a σ-vertex s_2 with the largest separating degree among σ-vertices other than s_1 is not massive or critical. Let $\delta = sd(s_1) - \lceil \frac{w(G)}{2} \rceil$. There are at least $2\delta + 2$ s_1-chains in 3-$blk(G)$. Let $\mathcal{M}$ be the set of s_1-chain leaves. By adding $2k, k \leq \delta$, edges to connect $2k+1$ vertices of $\mathcal{M}$, we reduce both $sd(s_1)$ and the weight of the graph by k.*

```
{* The input undirected graph G has at least 4 vertices; the algorithm finds a smallest
augmentation to triconnect G. *}
set of pairs of vertices function aug3(graph G);
    tree T2, T3;vertex s, y, z, u1, u2;
    L := ∅; T2 := 2-blk(G); T3 := 3-blk(G);
    let w(G) be the weight of G; {* Definition 5.3.5. *}
    while w(G) ≥ 2 and ∃ a c-vertex in T2 do
      if G is not connected then
1.      find y and z that are 3-block leaves or isolated 3-block vertices
        in different trees in T3
      else {* G is connected. *}
        let d be the largest separating degree among all σ-vertices in T3;
        if d > ⌈w(G)/2⌉ then {* ∃ an unique massive σ-vertex in T3 (Claim 5.4.10). *}
2.        let s be the massive σ-vertex in T3;
3.        find two s-chain leaves y and z in T3
        else {* d ≤ ⌈w(G)/2⌉ *}
4.        find two 3-block leaves y and z in T3 such that demanding vertices u and v
          of y and z, respectively, satisfy the condition given in Corollary 5.4.6
          {* Corollary 5.4.9 shows that this is always possible. *}
        fi
      fi;
      {* Claim 5.3.4 shows the following two statements will always succeed. *}
5.    find a demanding vertex u1 of y;
6.    find a demanding vertex u2 of z;
      add an edge between u1 and u2; L := L ∪ (u1, u2);
      update the 2-block graph T2; update the 3-block graph T3;
      if G is connected and d > ⌈w(G)/2⌉ then w(G) := w(G) − 1
      else {* G is not connected or d ≤ ⌈w(G)/2⌉. *} w(G) := w(G) − 2
      fi
    od;
    {* G is biconnected. *}
    return L ∪ aug2to3(G)
end aug3;
```

Algorithm 5.1: A linear time sequential algorithm for finding a smallest triconnectivity augmentation.

Proof. Let Tutte pairs corresponding to s_1 and s_2 be (a_1, a_2) and (a_3, a_4), respectively. Let G be a graph with h connected components. If there are at least two cutpoints in $\{a_i \mid 1 \leq i \leq 4\}$ and $|\{a_i \mid 1 \leq i \leq 4\}| = 4$, then the weight of the graph, $w(G)$, is at least $2(\sum_{i=1}^{4} d_2(a_i) + h - 7) + \sum_{i=1}^{2} d_3(s_i)$. We know that $sd(s_1) > \lceil \frac{w(G)}{2} \rceil$. If $sd(s_2) \geq \lceil \frac{w(G)}{2} \rceil$, then $\sum_{i=1}^{2} sd(s_i) > w(G)$. Since $sd(s_1) = d_3(s_1) + \sum_{i=1}^{2} d_2(a_i) + h - 4$ and $sd(s_2) = d_3(s_2) + \sum_{i=3}^{4} d_2(a_i) + h - 4$, $\sum_{i=1}^{4} d_2(a_i) < 6$. But $\sum_{i=1}^{4} d_2(a_i) \geq 6$ because there are at least two cutpoints in $\{a_i \mid 1 \leq i \leq 4\}$ and $|\{a_i \mid 1 \leq i \leq 4\}| = 4$. We have reached a contradiction. Thus s_2 is neither critical nor massive.

For the case of having less than two cutpoints in $\{a_i \mid 1 \leq i \leq 4\}$ and $|\{a_i \mid 1 \leq i \leq 4\}| = 4$, $w(G) \geq 2(\sum_{i=1}^{4} d_2(a_i) + h - 6) + \sum_{i=1}^{2} d_3(s_i)$ and $\sum_{i=1}^{4} d_2(a_i) \geq 4$. For the case of $|\{a_i \mid 1 \leq i \leq 4\}| < 4$, let $a_3 = a_4$; $w(G) \geq 2(\sum_{i=1}^{3} d_2(a_i) + h - 5) + \sum_{i=1}^{2} d_3(s_i)$ and $\sum_{i=1}^{2} d_2(a_i) \geq 2$. The claim can be proved in a similar way. For the size of the set of s_1-chain leaves and the rest of the claim, see Chapter 3 and [RG77]. □

Claim 5.4.11 *Let G be an undirected graph with at least 4 vertices. We can triconnect G by adding* $\max\{d, \lceil \frac{w(G)}{2} \rceil\}$ *edges using algorithm* `aug3`*, where d is the largest separating degree among all σ-vertices in 3-blk(G) and $w(G)$ is the weight of G.*

Proof. If G is not connected, algorithm `aug3` finds u_1 and u_2 in different connected components at steps 1, 5 and 6. From Lemma 5.4.8, we know that $\lceil \frac{w(G)}{2} \rceil$ is decreased by 1. From the definition of separating degree, d is decreased by 1 because the number of connected components is decreased by 1.

From Claim 5.4.10, we know that there can be at most one massive vertex s if $d > \lceil \frac{w(G)}{2} \rceil$. Algorithm `aug3` finds s at step 2. We reduce $\max\{d,$

$\lceil \frac{w(G)}{2} \rceil\}$ by 1 by adding a new edge between two s-chain leaves (step 3) if G is connected and unbalanced.

If G is balanced, algorithm `aug3` adds an edge (u_1, u_2) such that u_1 and u_2 are two demanding vertices that satisfy the condition given in Corollary 5.4.9. Thus $\lceil \frac{w(G)}{2} \rceil$ is decreased by 1 and G remains balanced. Algorithm `aug3` keeps doing this until G is biconnected. Notice that if G is biconnected, $w(G) = l_3(G)$. At this point, algorithm `aug3` calls algorithm `aug2to3` in Chapter 4 on the current biconnected graph. Hence the claim is true. □

Theorem 5.4.12

The triconnectivity augmentation number for a graph G equals $\max\{d, \lceil \frac{w(G)}{2} \rceil\}$, *where d is the largest separating degree among all σ-vertices in 3-blk(G) and $w(G)$ is the weight of G.*

Proof. By Lemma 5.3.6 and Claim 5.4.11. □

5.4.3 A Linear Time Implementation

Let s_G be the sum of the separating degrees of all Tutte pairs in the 3-block graph of a graph G with n vertices and m edges. By using techniques similar to those described in Chapters 3 and 4, algorithm `aug3` can be implemented to run in $O(n+m+s_G)$ time. Since s_G could be $\Omega(n^2)$ for a graph with n vertices, we do not have a linear time implementation for algorithm `aug3` if m is not $\Theta(n^2)$. However, Lemma 5.4.16 tells us of a possible method to obtain a linear time implementation (part of the claim in the lemma is also stated in [WN88]). Before the statement of Lemma 5.4.16, we need the following definition and corollary.

C-Component and S-Component

We now define components in a graph that are used in defining the weight of the graph.

Definition 5.4.13 *Let $s = (a_1, a_2)$ be a Tutte pair in $G = (V, E)$ and let $G' = G - \{a_1, a_2\}$. Let V_i' be the set of vertices of a connected component in G'. For each $c \in \{a_1, a_2\}$ that is a cutpoint, a* **c-component** *for s in G is an induced subgraph H on $V_i' \cup \{c\}$ such that $H - \{c\}$ does not contain any vertex adjacent to the vertex in $\{a_1, a_2\} - \{c\}$ in G. An* **s-component** *in G is an induced subgraph H on $V_i' \cup \{a_1, a_2\}$ such that there exists a vertex in $H - \{a_1, a_2\}$ adjacent to a_i in G, $i \in \{1, 2\}$.*

The following claim and corollary will show that each a-component and s-component "contributes" a certain value in the calculating of the weight.

Claim 5.4.14 *Let $s = \{a_1, a_2\}$ be a Tutte pair in a graph G and let $c \in \{a_1, a_2\}$. There exists a 2-block H in each c-component for s in G, where the corresponding b-vertex for H in 2-blk(G) is degree-1 and H is not in any other c-component or s-component. For each s-component in G, there exists either a 2-block H' whose corresponding b-vertex in 2-blk(G) is degree-1 or a 3-block H' whose corresponding β-vertex in 3-blk(G) is a 3-block leaf such that H' is not in any other c-component or s-component.*

Proof. See Appendix A. □

The following is a corollary of Claim 5.4.14.

Corollary 5.4.15 *Let $s = \{a_1, a_2\}$ be a Tutte pair in a graph G and let $c \in \{a_1, a_2\}$. Then each c-component for s in G contributes at least 2 in calculating $w(G)$ and each s-component in G contributes at least 1 in calculating $w(G)$.* □

We are now ready for Lemma 5.4.16.

Lemma 5.4.16 *Given a critical Tutte pair s_1 and another Tutte pair s_2, $s_1 \neq s_2$, in a connected graph G, let s_1 and s_2 share a common cutpoint c. Let $s_1 = (c, a_1)$ and let $s_2 = (c, a_2)$. Then $d_3(s_1) \geq d_3(s_2) + 2(d_2(a_2) - 1)$. If s_2 is also critical, then $d_2(a_1) = 1$ and $d_2(a_2) = 1$ (i.e., a_1 and a_2 are not cutpoints).*

Proof. The separating degree of s_i, $sd(s_i)$, is $d_3(s_i) + d_2(a_i) + d_2(c) - 3$, $i \in \{1, 2\}$.

There are at least $d_2(a_i) - 1$ a_i-components for each s_i in G such that they do not contain c. If s_1 and s_2 are in the same 2-block, there are at least $d_2(c) - 1$ c-components that do not contain a_1 and a_2, and there are at least $d_3(s_i) - 1$ s_i-components. The above components do not contain in each other. From Corollary 5.4.15, the weight of G is thus at least

$$\sum_{i=1}^{2} d_3(s_i) - 2 + 2(\sum_{i=1}^{2} d_2(a_i) - 2) + 2(d_2(c) - 1).$$

If s_1 and s_2 are in distinct 2-blocks, there are at least $d_2(c) - 2$ c-components that do not contain a_1 and a_2, and there are at least $d_3(s_i)$ s_i-components. Each of the above components is not contained in the other. From Corollary 5.4.15, the weight of G is at least

$$\sum_{i=1}^{2} d_3(s_i) + 2(\sum_{i=1}^{2} d_2(a_i) - 2) + 2(d_2(c) - 2).$$

Since s_1 is critical, $2sd(s_1) \geq w(G)$. By substituting the value of $sd(s_1)$ and the lower bound value of $w(G)$ into the above inequality, $d_3(s_1) \geq d_3(s_2) + 2(d_2(a_2) - 1)$.

If s_2 is also critical, then we can derive $d_3(s_2) \geq d_3(s_1) + 2(d_2(a_1) - 1)$ by an argument similar to the one given above. Thus $d_2(a_i) = 1$, for $i \in \{1, 2\}$.

□

Lemma 5.4.16 says that for a set of more than 1 critical σ-vertex whose corresponding Tutte pairs share a common cutpoint c, the other vertices in these Tutte pairs are not cutpoints. Let $\mathcal{S}_c$ be the set of Tutte pairs that share c and let $\mathcal{J}_c$ be the subset of $\mathcal{S}_c$ with the largest degree in 3-$blk(G)$ among all Tutte pairs in $\mathcal{S}_c$. A Tutte pair s in $\mathcal{S}_c$ is a *candidate* if and only if any one of the following conditions is true: (i) $\mathcal{J}_c = \{s\}$; (ii) $s \in \mathcal{J}_c$ and only the vertex c in each Tutte pair in $\mathcal{J}_c$ is a cutpoint; (iii) $s \in \mathcal{J}_c$ and both vertices in s are cutpoints. We can find a candidate for any $\mathcal{S}_c$. The following is a corollary of Lemma 5.4.16.

Corollary 5.4.17 *Let c be a cutpoint in G and let $\mathcal{S}_c$ be the set of Tutte pairs that share c. If $|\mathcal{S}_c| \geq 2$ and any one of the candidates is not critical, then none of the Tutte pairs in $\mathcal{S}_c$ is critical.*

Proof. Let $\mathcal{J}_c$ be the subset of $\mathcal{S}_c$ with the largest degree in 3-$blk(G)$ among all Tutte pairs in $\mathcal{S}_c$. From Lemma 5.4.16, we know that all critical Tutte pairs must be in $\mathcal{J}_c$. If there is only one candidate in $\mathcal{J}_c$, then it is the only one that can be critical. If the vertex other than c in each Tutte pair in $\mathcal{J}_c$ is not a cutpoint, then all Tutte pairs in $\mathcal{J}_c$ have the same separating degree. Thus if any one of the Tutte pair in $\mathcal{J}_c$ is not critical, then none of Tutte pairs in $\mathcal{S}_c$ is critical. If there are two distinct Tutte pairs $s_1 = (a_1, c)$ and $s_2 = (a_2, c)$ in $\mathcal{J}_c$ such that a_1 and a_2 are cutpoints, then none of them could be critical by Lemma 5.4.16. □

Let n be the number of vertices and m be the number of edges in the input graph. Using Lemma 5.4.16 and the *sorted table* data structure given in [RG77], we can implement algorithm `aug3` in $O(n+m)$ time using the following data structure.

Given a set $\mathcal{V}$ of p nodes with a key in each node whose value is $O(p)$, the *sorted table* is an array $\mathcal{L}$ of $O(p)$ entries. The ith entry of $\mathcal{L}$ points to a doubly linked list which contains all nodes in $\mathcal{V}$ with the key value i. For Tutte pairs that do not share any cutpoint with any other Tutte pairs, we maintain a sorted table using their separating degrees as keys. Entries in $\mathcal{L}$ are updated whenever their separating degrees are changed. The sum of the separating degrees of all σ-vertices in $\mathcal{L}$ is $O(n)$.

Given a sorted table $\mathcal{L}$, an equivalent *sorted list* is a doubly linked list on entries of $\mathcal{L}$ that do not point to an empty list. The relative order of entries in $\mathcal{L}$ is preserved. For each cutpoint c that is shared by Tutte pairs $\mathcal{S}_c = \{(c, a_1), \ldots, (c, a_r)\}$, where $r \geq 2$, we maintain the following information: y_c, a candidate for $\mathcal{S}_c$ (y_c is updated only if the current candidate no longer satisfies the condition of being a candidate) and $\mathcal{Q}_c$, a sorted list on the set of σ-vertices corresponding to Tutte pairs in $\mathcal{S}_c$ with their degrees in the 3-block graph as their key values. Note that we must re-choose y_c, where c is a cutpoint, if and only if the degree of any Tutte pair in $\mathcal{S}_c$ is changed or an endpoint of y_c is no longer a cutpoint. By maintaining the sorted list, y_c can be picked in constant time. We only pick candidates $O(n)$ times in total. Thus the above information can be maintained in a total of $O(n)$ time.

For a cutpoint c, let its y_c be (c, a_c). We maintain an entry with key value $sd(y_c)$ in the sorted table if and only if any one of the following conditions is true: (i) a_c is not a cutpoint; (ii) $|\mathcal{S}_{a_c}| \leq 1$; (iii) $|\mathcal{S}_{a_c}| > 1$ and $y_c = y_{a_c}$. This makes sure that if we update (decrease) the degree of a cutpoint, only a constant number of entries in the sorted table are updated. Note that if we have to change y_c for a cutpoint c, we only have to get a constant number of entries in and out the sorted table. Every entry in the sorted table is updated whenever its separating degree is changed.

During the augmentation using algorithm `aug3`, we create new σ-vertices by creating new polygons. The sum of the degrees (in the 3-block graph) of all created σ-vertices is $O(n)$. Since the sum of the degrees of all σ-vertices in the 3-block graph, the sum of the degrees of all c-vertices in the 2-block graph and the sum of the separating degrees of all σ-vertices in 3-$blk(G)$ are all $O(n)$, the total time to update the sorted table is $O(n)$. Using the above method, we can implement algorithm `aug3` in linear time.

Claim 5.4.18 *Algorithm* `aug3` *runs in linear time.*

Proof. The 2-block graph and the 3-block graph can be obtained in linear time by using procedures in [Ram93]. Using implementation techniques similar to those given in Chapter 4, algorithm `aug3` can be implemented to run in time $O(n+m)$ plus the time to maintain the separating degrees of all σ-vertices in the 3-block graph. A naive implementation for maintaining separating degrees runs in $O(n^2)$ time. Using our sorted table and sorted lists, the time for maintaining all separating degrees is $O(n)$.

We now specify the time spent in updating the 2-block graph and the 3-block graph. Note that we do not have to update 2-$blk(G)$ and 3-$blk(G)$ if G is not connected. Without lose of generality, let G be connected, but not biconnected. In the following discussion, we use definitions given in Page 110 (before the statement of Lemma 5.4.2). If $|\Im_1| = 1$, then 2-$blk(G)$ is a path (Lemma 5.4.3). If $|\Im_1| = 2$, then 2-$blk(G)$ contains only two cutpoints and the next edge we add will biconnect G (Lemma 5.4.2). Thus the total time for updating 2-$blk(G)$ for $|\Im_1| > 0$ is linear. If $|\Im_2| = 2$, then it is also true that the next edge we add will biconnect G (Lemma 5.4.4). For the case of $|\Im_2| < 2$, we first root 2-$blk(G)$ at a b-vertex with degree ≥ 2. It is obvious that we can

always make sure that the implied path in 2-$blk(G)$ between the two endpoints of each new edge added passes the root. Thus the overall updating time is also linear for $|\Im_2| < 2$.

For updating 3-$blk(G)$, we first root each tree in 3-$blk(G)$ at a β-vertex with degree ≥ 2. We do not have to worry about the case when $|\Im_1| = 2$ or $|\Im_2| = 2$. If $|\Im_2| < 2$, we can make sure that each path in the collection of implied path in 3-$blk(G)$ between the two endpoints of each new edge added passes the root of the tree where it is in. If $|\Im_1| = 1$, each tree in the current 3-$blk(G)$ will be passed at most twice before the graph becomes biconnected. Let P be the implied path in 3-$blk(G)$ between two endpoints of an new added edge. We can afford to reroot each tree in 3-$blk(G)$ a constant number of times to make sure that each P passes through the root of the tree (in 3-$blk(G)$) that contains P. Thus the overall updating time for 3-$blk(G)$ is also linear. Thus the overall time complexity is $O(n + m)$ if we use a data structure that is the same as the one used in Chapter 4 and [RG77] for representing 2-$blk(G)$ and 3-$blk(G)$. □

5.5 An Efficient Parallel Algorithm

In this section, we describe an efficient algorithm for finding a smallest triconnectivity augmentation for a graph that is not biconnected. The algorithm first examines the input graph. If it is not connected, we connect it using r_1 edges such that the triconnectivity augmentation number decreases by r_1. Then we examine whether it is biconnected. If it is not biconnected, we biconnect it using r_2 edges such that the triconnectivity augmentation number decreases by r_2. We then apply the algorithm developed in Chapter 4 to optimally triconnect it.

The structure of this section is as follows. We first describe properties of the 3-block graph for an input graph that is not biconnected. We then describe a simple parallel algorithm for connecting the input graph. Finally, we develop an algorithm for biconnecting the current connected graph.

5.5.1 Properties of the 3-Block Graph

We first describe a lemma for properties of the 3-block graph which will be used in designing our parallel algorithm.

Lemma 5.5.1 *Given a graph G, let s_x, s_y and s_z be three distinct separating pairs in 3-blk(G) with $sd(s_x) \geq sd(s_y) \geq sd(s_z)$. Recall that $w(G)$ is the weight of the graph defined in Definition 5.3.5.*
(i) If s_x, s_y and s_z do not share a common cutpoint, then $sd(s_z) \leq \frac{w(G)+14}{3}$.
(ii) If s_x, s_y and s_z share a common cutpoint c, then $sd(s_z) - d_2(c) \leq \frac{w(G)-2}{3}$.

Proof. We prove the first part of the claim by case analysis. Let $s_i = (c_{i,1}, c_{i,2})$, $i \in \{x, y, z\}$. It is possible that a vertex in a separating pair in $\{s_x, s_y, s_z\}$ is also in another separating pair in $\{s_x, s_y, s_z\}$. Let $\{c_{x,1}, c_{x,2}, c_{y,1}, c_{y,2}, c_{z,1}, c_{z,2}\} = \{c_i | 1 \leq i \leq r\}$, where $r \leq 6$. Thus $w(G) \geq d_3(s_x) + d_3(s_y) + d_3(s_z) - 2 - 2\sum_{i=1}^{r}(d_2(c_i) - 2)$. Recall that $sd(s_i) = d_3(s_i) + d_2(c_{i,1}) + d_2(c_{i,2}) - 4$, for $i \in \{x, y, z\}$. Since $r \leq 6$, $w(G) \geq sd(s_x) + sd(s_y) + sd(s_z) - 14$. We know that $sd(s_z) \leq sd(s_y) \leq sd(s_x)$. Thus $3sd(s_z) \leq sd(s_x) + sd(s_y) + sd(s_z)$. Hence $3sd(s_z) \leq w(G) + 14$.

The second part of the claim can be proved in a similar way. We know that $r = 4$ in this case. Let $c = c_4$ be the endpoint that is shared by all three separating pairs. Thus

$$w(G) \geq d_3(s_x) + d_3(s_y) + d_3(s_z) - 2 + 2(\sum_{i=1}^{3}(d_2(c_i) - 1) + d_2(c) - 1).$$

Hence

$$\begin{aligned} w(G) &\geq sd(s_x) + sd(s_y) + sd(s_z) - d_2(c) + 2 \\ &\geq 3(sd(s_z) - d_2(c)) + 2. \end{aligned}$$

This implies $3(sd(s_z) - d_2(c)) \leq w(G) - 2$. □

The following fact about the properties on the structure of the 2-block graph and the 3-block graph is well-known.

Fact 5.5.2 *Given an undirected graph G and its 2-block graph 2-blk(G), let b be a 2-block leaf (or an isolated b-vertex) in 2-blk(G) and let H_b be its corresponding 2-block in G. Then one of the following statements is true.*
(i) H_b contains two distinct 3-block leaves.
(ii) H_b is a vertex of degree 1.
(iii) H_b contains only one 3-block leaf.
(iv) H_b corresponds to an isolated 3-block vertex.

Using the above fact, we want to find vertices in a 2-block leaf such that if we add an edge between two such vertices in different 2-block leaves, the number of 2-block leaves and the number of 3-block leaves both reduce by 2. Thus if we add an edge properly, we can reduce the triconnectivity number and biconnect the input graph at the same time.

Definition 5.5.3 *Let G be an undirected graph. The* **two demanding vertices** *$q_1(b)$ and $q_2(b)$ of a 2-block leaf (or an isolated b-vertex) b in 2-blk(G) (not necessary to be distinct), are defined as follows. Let H_b be the subgraph of G represented by b.*
(i) If there are two distinct 3-block leaves t_1 and t_2 in the 3-block tree 3-blk(H_b),

then let $q_1(b)$ be a demanding vertex of t_1 and let $q_2(b)$ be a demanding vertex of t_2.

(ii) If H_b consists of a single vertex v, then let $q_1(b) = v$ and let $q_2(b) = v$.

(iii) If there is only one 3-block leaf t in 3-$blk(H_b)$, then let $q_1(b)$ and $q_2(b)$ be two distinct demanding vertices of t.

(iv) If H_b is triconnected, then let $q_1(b)$ and $q_2(b)$ be two distinct demanding vertices in H_b.

The following claim can be easily verified from the definition of $w(G)$.

Claim 5.5.4 *Given a graph G with two 2-block leaves b_1 and b_2 in 2-$blk(G)$, let $q_1(b_i)$ be one of the two demanding vertices of b_i, $i \in \{1, 2\}$. Then $w(G \cup \{(q_1(b_1), q_1(b_2))\}) = w(G) - 2$.* □

5.5.2 The Graph is not Connected

If the input graph is not connected, let the set of connected components be $\{G_1, G_2, \ldots, G_h\}$. Let b_i be a 2-block leaf (or an isolated b-vertex) in 2-$blk(G_i)$, $1 \leq i \leq h$. We find the two demanding vertices $q_1(b_i)$ and $q_2(b_i)$ for each b_i, $1 \leq i \leq h$. The algorithm adds a set of $h-1$ edges $\mathcal{A}$, where $\mathcal{A}$ equals $\{(q_2(b_1), q_1(b_2)), (q_2(b_2), q_1(b_3)), \ldots, (q_2(b_{h-1}), q_1(b_h))\}$.

Claim 5.5.5 *Let the original graph be G and let the graph obtained by adding the above $h-1$ edges be G'. The graph G' is connected and the triconnectivity augmentation number decreases by $h-1$.*

Proof. It is obvious that G' is connected. By adding $\mathcal{A}$, we create some separating pairs, but not Tutte pairs. Thus the largest degree among all σ-vertices in 3-$blk(G)$ is the same with the largest degree among all σ-vertices

```
set of pairs of vertices function p3aug0to1(forest T2);
{* T2 is the 2-block graph for the input graph. *}
    set of pairs of vertices L;
    let T2,i be the ith tree in T2, 1 ≤ i ≤ h;
    L := ∅; {* L is the set of matched pairs. *}
    let bi be a leaf or an isolated vertex in T2,i, 1 ≤ i ≤ h;
    let q1(bi) and q2(bi) be the two demanding vertices for bi, 1 ≤ i ≤ h;
    pfor i = 1 .. h − 1 do
        L := L ∪ {(q2(bi), q1(bi+1))}
    rofp;
    return L
end p3aug0to1;
```

Algorithm 5.2: Parallel algorithm to connect a graph such that the set of edges added is a subset of a smallest triconnectivity augmentation.

in 3-$blk(G')$. It is easy to see that that $w(G') = w(G) - 2(h-1)$ from the definition of the weight of the graph. Thus the triconnectivity augmentation number decreases by $h-1$. □

5.5.3 The Graph is Connected, but not Biconnected

In this section, we consider the case when the input graph is balanced and connected. If the graph is not balanced, we can reduce the triconnectivity augmentation number by k by adding k edges and obtain a balanced graph using a method similar to the one used in biconnectivity augmentation as described in Chapter 3.

Let s_1 be a σ-vertex with the largest separating degree in 3-$blk(G)$. Depending on the value of $sd(s_1)$, we have two subcases for this problem. If $sd(s_1) \leq \frac{w(G)}{4}$, we are free to add up to $\lfloor \frac{w(G)}{2} \rfloor - sd(s_1)$ edges without worrying about whether any σ-vertex will become massive. We only have to make sure that each time we add an edge, the two endpoints of the new edge satisfy the leaf-connecting condition (Lemma 5.4.8). We can reduce the triconnectivity augmentation number by exactly the number of new edges added. If $sd(s_1) >$

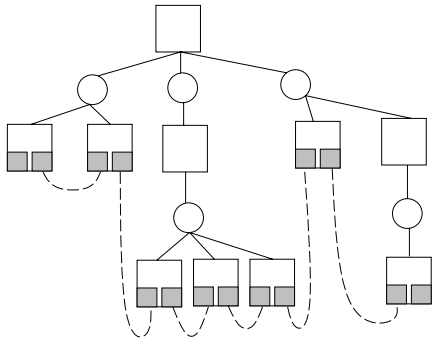


Figure 5.6: Illustrating case 1 for biconnecting a connected graph while at the same time reducing the triconnectivity augmentation number by the number of edges added. The 2-block tree of the current graph is shown with a circle representing a c-vertex and a rectangle representing a b-vertex. The two shadowed rectangles in each leaf are the two demanding vertices of each 2-block leaf. We add edges between demanding vertices of adjacent 2-block leaves.

$\frac{w(G)}{4}$, we have to add new edges carefully in order not to unbalance the resulting graph. In each subcase, we make sure the number of edges added is either a constant fraction of the current triconnectivity augmentation number or the graph after the addition of new edges is biconnected. Thus we only have to apply this algorithm $O(\log n)$ times to biconnect the input graph.

We now describe the two cases in detail.

Case 1: $sd(s_1) \leq \frac{w(G)}{4}$

In this case, we match demanding vertices between 2-block leaves. Since the path between each edge added passes through at least a cutpoint, we reduce the weight of the graph by 2 by adding an edge (Lemma 5.4.8). We add at most $\lfloor \frac{w(G)}{2} \rfloor - sd(s_1)$ edges. Thus no σ-vertex becomes massive. An example is shown in Figure 5.6.

We first describe the algorithm in Algorithm 5.3. Then we prove its correctness.

Claim 5.5.6 *Let the G be the current graph and let G' be the graph obtained*

```
set of pairs of vertices function case1(forest T2, T3);
{* T2 is the 2-block graph; T3 is the 3-block graph *}
    set of pairs of vertices L;
    let s1 be a σ-vertex with the largest separating degree in T3;
    let l2 be the number of 2-block leaves in T2;
    k := min{l2 − 1, ⌊w(G)/2⌋ − sd(s1)};
    let b1, b2, ...,bk+1 be k + 1 2-block leaves in T2;
    let q1(bi) and q2(bi) be the two demanding vertices of bi, 1 ≤ i ≤ k + 1;
    L := ∅; {* L is the set of matched pairs. *}
    pfor i = 1 .. k do
        L := L ∪ {(q2(bi), q1(bi+1))}
    rofp;
    return L
end case1;
```

Algorithm 5.3: Parallel algorithm for case 1 of optimally triconnecting a connected graph.

from G by adding an edge between each pair of matched vertices found in function `case1`. *Then either G' is biconnected or the number of edges added is* $\geq \frac{w(G)}{4}$.

Proof. We add exactly $k = \min\{l_2 - 1, \lfloor \frac{w(G)}{2} \rfloor - sd(s_1)\}$ edges. Since $sd(s_1) \leq \frac{w(G)}{4}$, the claim is true if $k = \lfloor \frac{w(G)}{2} \rfloor - sd(s_1)$. If $k = l_2 - 1$, we will prove in the following paragraph that there is no cutpoint in G'. First we observe that if c is a cutpoint in G', then it is a cutpoint in G. Any cutpoint in G is no longer a cutpoint in G', since there is a path in 2-$blk(G')$ from any leaf in 2-$blk(G)$ to any other leaf in 2-$blk(G)$ that does not pass through any cutpoint in G. □

Claim 5.5.7 *Let G be the current balanced graph and let G' be the graph obtained from G by adding an edge between each pair of vertices found in function* `case1`. *$A_3(G') = A_3(G) - k$ and G' remains balanced.*

Proof. We first observe that by adding edges in a connected graph G, we do not increase the largest separating degree among all σ-vertices in 3-$blk(G)$. Let the

edges be added in the sequence from $(q_2(b_1), q_1(b_2))$ to $(q_2(b_k), q_1(b_{k+1}))$. Let the graph obtained from G by adding the first i edges be G_i. Since the path between $q_2(b_{i+1})$ and $q_1(b_{i+2})$ passes though a cutpoint in 2-$blk(G_i)$, $w(G_{i+1}) = w(G_i)-2$ (Lemma 5.4.8). □

Case 2: $sd(s_1) > \frac{w(G)}{4}$

Let s_1, s_2 and s_3 be three σ-vertices in 3-$blk(G)$ such that no other σ-vertex has a larger separating degree than that of s_i, $1 \leq i \leq 3$, and $sd(s_1) \geq sd(s_2) \geq sd(s_3)$. From Lemma 5.5.1, we know that no σ-vertex will become critical or massive if we add at most $\frac{w(G)}{2} - sd(s_3)$ edges such that the path between the two endpoints of each edge added passes though s_1 and s_2 and the cutpoint c, if any, which is shared by s_1 and s_2. We also make sure each time we add an edge, the weight of the graph decreases by 2. Thus the triconnectivity augmentation number decreases by the number of edges added. Our algorithm will try to "get rid" of as many cutpoints as possible.

There are four phases for this part of the algorithm. Let c be the cutpoint, if any, that is shared by s_1 and s_2. In phase 1, we first make sure that s_1 and s_2 are in the same biconnected component. We then match demanding vertices between c-components such that c is no longer a cutpoint in the resulting graph obtained by adding an edge between each matched pair. Let $s_1 = (c_{1,1}, c_{1,2})$ and let $s_2 = (c_{2,1}, c_{2,2})$. Let W_i, $i \in \{1, 2\}$, be the set of $c_{i,1}$- and $c_{i,2}$-components that do not contain s_i. Let Y_i, $i \in \{1, 2\}$, be the set of s_i-components that contain a cutpoint and let Y_i' be the set of s_i-components that do not contain a cutpoint. In phase 2, we match demanding vertices of elements in $W_1 \cup Y_1 \cup Y_1'$ and demanding vertices of elements in $W_2 \cup Y_2 \cup Y_2'$. For each pair of vertices we match, the path between them in the 2-block graph passes through at least one cutpoint and the path between them in the 3-bock

graph passes through s_1 and s_2. In phase 3, we match demanding vertices of elements in Y_1' and demanding vertices of elements in Y_2' such that the separating degree of s_2 becomes 2 after adding an edge between the two vertices in each matched pair. We also match among demanding vertices of elements in W_1. Finally in phase 4, we match demanding vertices of $Y_1 \cup Y_1'$ with demanding vertices of 2-block leaves. By doing this, we reduce the separating degree of s_1 to 2 if there are enough 2-block leaves left; otherwise, the graph is biconnected. We now describe the fours phases in detail.

Phase 1: Let c be the common cutpoint shared by s_1 and s_2. If s_1 and s_2 are not in the same biconnected component, we match a demanding vertex in an s_1-component and a demanding vertex in an s_2-component. After adding an edge between the pair of demanding vertices, s_1 and s_2 are in the same biconnected component. We then find the two demanding vertices $q_1(b_i)$ and $q_2(b_i)$ for the rest of c-components in $\{C_i \mid 1 \leq i \leq r\}$, where b_i is a 2-block leaf in C_i. We match $q_2(b_i)$ with $q_1(b_{i+1})$, $1 \leq i < r$, and match $q_2(b_r)$ with a demanding vertex in an s_1-component. An example is shown in Figure 5.7.

Phase 2: Recall that $s_1 = (c_{1,1}, c_{1,2})$ and $s_2 = (c_{2,1}, c_{2,2})$. Let W_i be the set of $c_{i,1}$- and $c_{i,2}$-components that do not contain s_i, $i \in \{1, 2\}$. Recall also that Y_i is the set of s_i-components that contain a cutpoint, $i \in \{1, 2\}$.

We try to eliminate cutpoints from endpoints of s_1 and s_2 by matching demanding vertices of components in $W_1 \cup Y_1 \cup Y_1'$ and those in $W_2 \cup Y_2 \cup Y_2'$. By doing this, we reduce separating degrees of both s_1 and s_2 and also decrease the sum of degrees of all c-vertices in the 2-block tree. An example is shown in Figure 5.8.

Phase 3: Recall that Y_i' is the set of s_i-components that do not contain a cutpoint, $i \in \{1, 2\}$. Note that after phase 2, $W_2 = \emptyset$ and $Y_2 = \emptyset$. Note also

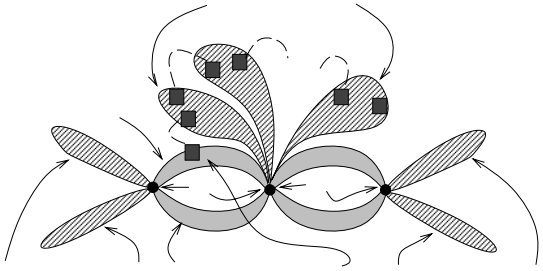


Figure 5.7: Illustrating phase 1 of case 2 for biconnecting a connected graph while at the same time reducing the triconnectivity augmentation number by the number of edges added. The current graph is shown with two separating pairs $s_1 = (c_1, c)$ and $s_2 = (c_2, c)$ sharing a common cutpoint c. The two rectangles shown inside each c-component C are the two demanding vertices of a 2-block leaf in C. The rectangle shown inside an s_1-component h is a demanding vertex of a 3-block leaf in h. We merge c-components with h by adding edges between demanding vertices properly. After adding the set of edges, vertex x becomes the demanding vertex for the new s_1-component formed.

```
set of pairs of vertices function phase1(vertex s1, s2;
modifies set of components Z1, Z2; modifies forest T2, T3);
{* s1 and s2 are two σ-vertices with the largest separating degrees. *}
{* Zi is the set of si-components, i ∈ {1,2}. *}
{* s1 and s2 share a common cutpoint c. *}
   set of pairs of vertices L;
   L := ∅; {* L is the set of matched pairs. *}
   if s1 and s2 are not in the same biconnected component then
      let zi ∈ Zi, i ∈ {1,2}; q(zi) := a demanding vertex of a 3-block leaf in zi;
      L := L ∪ {(q(z1), q(z2))}; Zi := Zi − {zi}, i ∈ {1,2}
   fi;
   let {Ci | 1 ≤ i ≤ r} be the set of c-components that do not contain s1 and s2;
   pfor i = 1 .. r do
      let bi be a 2-block leaf in Ci;
      let q1(bi) and q2(bi) be the two demanding vertices of bi;
      L := L ∪ {(q2(bi), q1(bi+1))}
   rofp;
   let h ∈ Z1; find a 3-block leaf t in h;
   let v be the associated demanding vertex of t;
   L := L ∪ {(q2(br), v)};
   replace the associated demanding vertex of t with q1(b1);
   return L
end phase1;
```

Algorithm 5.4: Parallel algorithm for phase 1 of case 2 for optimally triconnecting a connected graph.

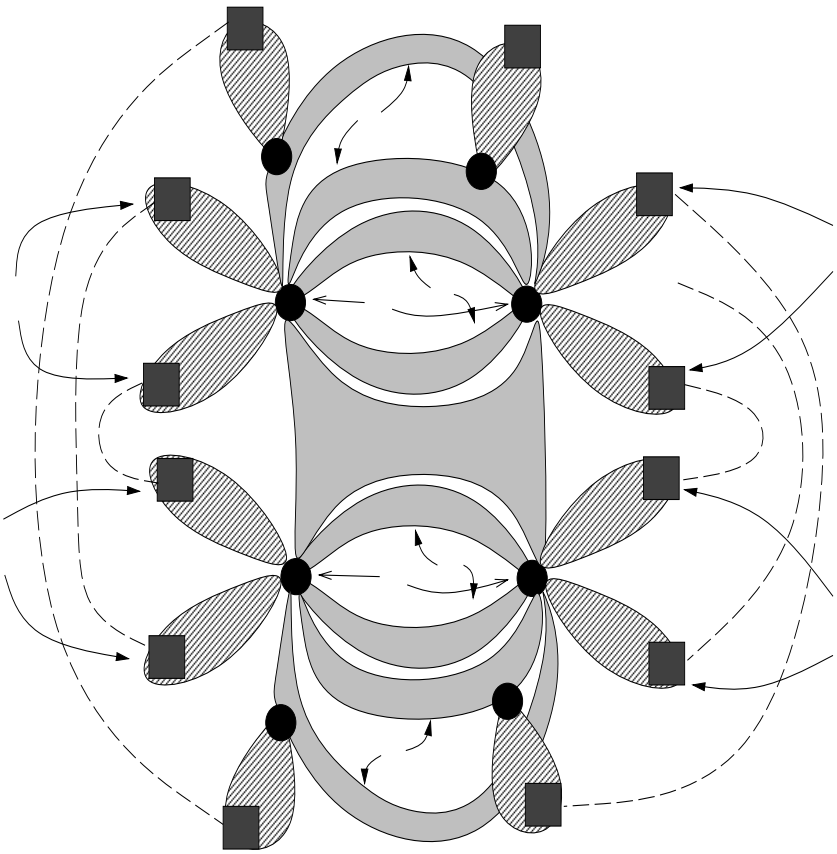


Figure 5.8: Illustrating phase 2 of case 2 for biconnecting a connected graph while at the same time reducing the triconnectivity augmentation number by the number of edges added. The current graph is shown with two separating pairs $s_1 = (c_{1,1}, c_{1,2})$ and $s_2 = (c_{2,1}, c_{2,2})$. The set W_i consists of $c_{i,1}$- and $c_{i,2}$-components, $i \in \{1, 2\}$. The set Y_i consists of the s_i-components that contain a cutpoint, $i \in \{1, 2\}$. The set Y_i' consists of s_i-components that do not contain a cutpoint, $i \in \{1, 2\}$. We add edges such that the path in 2-$blk(G)$ between the two endpoints of each added edge passes through at least a cutpoint.

```
set of pairs of vertices function phase2(modifies set of components W_1, W_2,
Y_1, Y_2, Y_1', Y_2'; modifies forest T_2, T_3);
{* W_i is the set of c_{i,1}- and c_{i,2}-components, i ∈ {1,2}. *}
{* Y_i is the set of s_i-components that contain a cutpoint, i ∈ {1,2}. *}
{* Y_i' is the set of s_i-components that do not contain a cutpoint, i ∈ {1,2}. *}
   set of pairs of vertices L;
   L := ∅; {* L is the set of matched pairs. *}
   Z_i := W_i ∪ Y_i ∪ Y_i', i ∈ {1,2};
   number elements in Z_i from 1 to |Z_i| such that elements in W_i are numbered before
   elements in Y_i and elements in Y_i are numbered before elements in Y_i', i ∈ {1,2};
   k := min{|Z_2| − 2, max{|W_1 ∪ Y_1|, |W_2 ∪ Y_2|}};
   pfor i = 1 .. k do
      let z_{1,i} be the ith element of Z_1; let z_{2,i} be the ith element of Z_2;
      let a_i be a 2-block leaf in z_{1,i}; let b_i be a 2-block leaf in z_{2,i};
      let q_1(a_i) be be one of the two demanding vertices of a_i;
      let q_1(b_i) be be one of the two demanding vertices of b_i;
      L := L ∪ (q_1(a_i), q_1(b_i));
      if i ≤ |W_1| then remove z_{1,i} from W_1
      else if |W_1| < i ≤ |W_1| + |Y_1| then remove z_{1,i} from Y_1
      else if |W_1| + |Y_1| < i then remove z_{1,i} from Y_1'
      fi;
      if i ≤ |W_2| then remove z_{2,i} from W_2
      else if |W_2| < i ≤ |W_2| + |Y_2| then remove z_{2,i} from Y_2
      else if |W_2| + |Y_2| < i then remove z_{2,i} from Y_2'
      fi
   rofp;
   return L
end phase2;
```

Algorithm 5.5: Parallel algorithm for phase 2 of case 2 to optimally triconnect a connected graph.

that $W_1 = \emptyset$ and $Y_1 = \emptyset$ if $|Y_2'| > 2$. If $|Y_2'| > 2$, we match demanding vertices of all but two elements in Y_2' with demanding vertices of elements in Y_1'. If $|Y_2'| = 2$, then we match between demanding vertices of elements in W_1. An example is shown in Figure 5.9.

Phase 4: Note that after phase 3, $|Y_2'| = 2$, $W_1 = \emptyset$, $W_2 = \emptyset$, and $Y_2 = \emptyset$. We match demanding vertices of elements in Y_1 with demanding vertices of elements in Y_1'. If $Y_1 \neq \emptyset$ after the previous matching, we match the rest of them within themselves. If $Y_1' \neq \emptyset$ after the matching, we match them

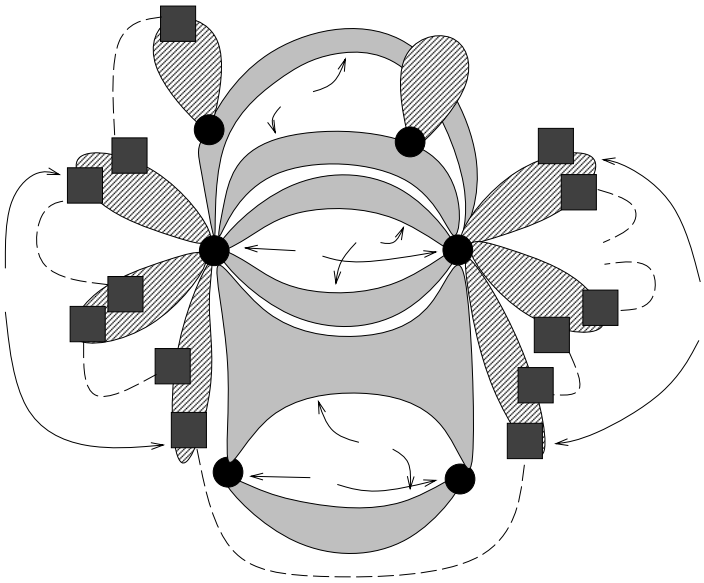


Figure 5.9: Illustrating phase 3 of case 2 for biconnecting a connected graph while at the same time reducing the triconnectivity augmentation number by the number of edges added. The current graph is shown with two separating pairs s_1 and s_2. We add edges such that the separating degree of s_2 in the resulting is at most 2 and the two vertices in s_1 are no longer cutpoints.

with demanding vertices of the remaining 2-block leaves. An example is shown in Figure 5.10.

We now describe the complete algorithm for case 2 in Algorithm 5.8.

The following claim can easily be verified.

Claim 5.5.8 *Let $G' = G \cup \{(u_1, u_2)\}$, where u_1 and u_2 are vertices in G, but $(u_1, u_2) \notin G$. For every Tutte pair s in G', the following is true.*
(i) If s is also a Tutte pair in G, then $sd(s)$ in G' is less than or equal to $sd(s)$ in G.
(ii) if s is not a Tutte pair in G, then $sd(s)$ in G' equals 2. □

Claim 5.5.9 *Let G' be the graph obtained by adding an edge between each pair*

```
set of pairs of vertices function phase3(modifies set of components W_1, Y_1,
Y'_1, Y'_2; modifies forest T_2, T_3);
{* W_1 is the set of c_{1,1}- and c_{1,2}-components. *}
{* Y_1 is the set of s_1-components that contain a cutpoint. *}
{* Y'_i is the set of s_2-components that do not contain a cutpoint, i ∈ {1,2}. *}
   set of pairs of vertices L;
   L := ∅; {* L is the set of matched pairs. *}
   if |Y'_2| > 2 then
      pfor i = 1 .. |Y'_2| − 2 do
         let z_{2,i} be the ith element of Y'_2; let z_{1,i} be the ith element of Y'_1;
         let a_i be a 3-block leaf in z_{2,i}; let b_i be a 3-block leaf in z_{1,i};
         let q(a_i) be a demanding vertex of a_i; let q(b_i) be a demanding vertex of b_i;
         L := L ∪ {(q(a_i), q(b_i))};
         remove z_{2,i} from Y'_2; remove z_{1,i} from Y'_1
      rofp
   else {* |Y'_2| ≤ 2 *}
      k := |W_1| − 1;
      pfor i = 1 .. k do
         let z_i be the ith element of W_1;
         let a_i be a 2-block leaf in z_i;
         let q_1(a_i) and q_2(a_i) be the two demanding vertices of a_i;
         L := L ∪ {(q_2(a_i), q_1(a_{i+1}))}
      rofp;
      let t be a 3-block leaf in Y_1 ∪ Y'_1; let q(t) be a demanding vertex of t;
      L := L ∪ {(q_2(a_{|W_1|}), q(t))};
      replace the demanding vertex of t with q_1(a_1)
   fi;
   return L
end phase3;
```

Algorithm 5.6: Parallel algorithm for phase 3 of case 2 to optimally triconnect a connected graph.

of matched vertices. It is either the case that G' is biconnected or the number of pairs matched in case 2 is at least $\frac{W(G)}{8}$ if $w(G) \geq 56$.

Proof. If we cannot find enough 2-block leaves to be matched in phase 4. That means for each 2-block leaf b in 2-$blk(G)$, there is an edge from b to the 2-block contains s_1 and s_2. Thus G' is biconnected. Otherwise, either we add at least $\frac{w(G)-14}{6}$ edges or we reduce the separating degree of s_1 to 2. Since each time we add an edge, the separating degree of s_1 reduces by at most 1 and $sd(s_1) > \frac{w(G)}{4}$. Thus we add at least $\min\{\frac{w(G)-14}{6}, \frac{w(G)}{4} - 2\}$ edges. Since

```
set of pairs of vertices function phase4(modifies set of components Y_1, Y_1';
modifies forest T_2, T_3);
{* Y_1 is the set of s_1-components that contain a cutpoint. *}
{* Y_1' is the set of s_1-components that do not contain a cutpoint. *}
   set of pairs of vertices L;
   L := ∅; {* L is the set of matched pairs. *}
   if Y_1 ≠ ∅ and |Y_1 ∪ Y_1'| > 2 then
      k := min{|Y_1 ∪ Y_1'| − 2, |Y_1|};
      pfor i = 1 .. ⌊k/2⌋ do
         let z_i be the ith element of Y_1;
         let a_i be a 2-block leaf in z_i;
         let q_1(a_i) and q_2(a_i) be the two demanding vertices of a_i;
         L := L ∪ {(q_1(a_{2i−1}), q_1(a_{2i}))};
         remove z_{2i−1} and z_{2i} from Y_1
      rofp
   fi;
   if |Y_1 ∪ Y_1'| > 2 then
      Y := Y_1 ∪ Y_1';
      let l_2 be the number of 2-block leaves that do not contain a matched vertex;
      k := min{l_2, |Y| − 2};
      pfor i = 1 .. k do
         let y_i be the ith element in Y;
         let a_i be the a 3-block leaf in y_i;
         let b_i be the ith 2-block leaf that does not contain a matched vertex;
         let q(a_i) be a demanding vertex of a_i;
         let q_1(b_i) be one of the two demanding vertices of b_i;
         L := L ∪ {(q(a_i), q_1(b_i))}
      rofp
   fi;
   return L
end phase4;
```

Algorithm 5.7: Parallel algorithm for phase 4 of case 2 for optimally triconnecting a connected graph.

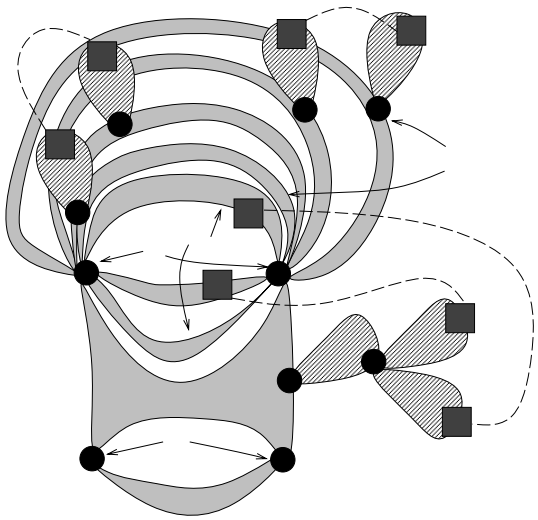


Figure 5.10: Illustrating phase 4 of case 2 for biconnecting a connected graph while at the same time reducing the triconnectivity augmentation number by the number of edges added. The current graph is shown with two separating pairs s_1 and s_2. We match among demanding vertices in s_1-components that contain a cutpoint. Then we match demanding vertices between the rest of s_1-components and demanding vertices of 2-block leaves in the current graph. After this phase, either the resulting graph is biconnected or the separating degrees of s_1 and s_2 equal to 2.

$\frac{w(G)-14}{6} \geq \frac{w(G)}{8}$ if $w(G) \geq 56$ and $\frac{w(G)}{4} - 2 \geq \frac{w(G)}{8}$ if $w(G) \geq 16$, the number of matched pairs is at least $\frac{w(G)}{8}$ if $w(G) \geq 56$. □

Claim 5.5.10 *Let G be the input balanced graph to function* `case2` *and let G' be the graph obtained from G by adding an edge between two endpoints of each matched pair. Let k be the number of edges added. $A_3(G') = A_3(G) - k$ and G' remains balanced.*

Proof. We first prove that each time we add an edge between a pair of matched vertices, the separating degree of s_i reduces by 1 if $sd(s_i) \geq 3$, $i \in \{1, 2\}$. Each edge added between a pair of matched vertices found in phase 1 reduces the separating degree of both s_1 and s_2 by 1, since two c-components are merged

set of pairs of vertices function `case2`(**forest** T_2, T_3);
{$*$ T_2 is the 2-block graph; T_3 is the 3-block graph. $*$}
  set of pairs of vertices L;
  $L := \emptyset$; {$*$ L is the set of matched pairs. $*$}
  let s_1, s_2 and s_3 be three σ-vertices in T_3 such that no other σ-vertex has larger separating degree than theirs and $sd(s_1) \geq sd(s_2) \geq sd(s_3)$;
  $maxE := \frac{w(G)}{2} - sd(s_3)$; {$*$ maximum number of edges to be added $*$}
  let $s_1 = (c_{1,1}, c_{1,2})$; let $s_2 = (c_{2,1}, c_{2,2})$;
  $Z_i :=$ the set of s_i-components, $i \in \{1,2\}$;
  $L :=$ `phase1`(s_1, s_2, Z_1, Z_2, T_2, T_3);
  $Z_i := Y_i \cup Y_i'$ such that Y_i are elements in Z_i that contains a cutpoint, $i \in \{1,2\}$;
  $W_i :=$ the set of $c_{i,1}$- and $c_{i,2}$-components, $i \in \{1,2\}$;
  $L := L \cup$ `phase2`(W_1, W_2, Y_1, Y_2, Y_1', Y_2', T_2, T_3);
  $L := L \cup$ `phase3`(W_1, Y_1, Y_1', Y_2', T_2, T_3);
  $L := L \cup$ `phase4`(Y_1, Y_1', T_2, T_3);
  if $maxE < |L|$ **then** remove the last $|L| - maxE$ pairs added from L **fi**;
  return L
end `case2`;

Algorithm 5.8: Complete parallel algorithm for case 2 of optimally triconnecting a connected graph.

into one c-component where c is a cutpoint shared by s_1 and s_2. In phases 2 to 4, the path in 3-$blk(G)$ between the pair of matched vertices passes through both s_1 and s_2. Thus the separating degree of s_i reduces by 1 each time we add an edge if $sd(s_i) \geq 3$, $i \in \{1,2\}$.

We now prove that each time we add an edge, the weight of the graph decreases by 2. Notice that the path between pairs of vertices found in phases 1 to 3 passes through at least a cutpoint in the 2-block graph. Thus the weight of the graph decreases by 2 each time we add an edge between the pair of matched vertices (Lemma 5.4.8). In phase 4, the path between each pair of matched vertices passes through s_1 and s_2 where $d_3(s_i) \geq 3$, $i \in \{1,2\}$. Thus each matched pair satisfies the leaf-connecting condition. □

```
set of pairs of vertices function p3aug1to2(forest T2, T3);
{* The current graph is connected, but not biconnected. *}
{* T2 is the current 2-block graph and T3 is the current 3-block graph. *}
    let s1 be a σ-vertex in 3-blk(G) with the largest separating degree;
    if sd(s1) ≤ w(G)/4 then return case1(T2, T3)
    else {* sd(s1) > w(G)/4 *} return case2(T2, T3)
    fi
end p3aug1to2;
```

Algorithm 5.9: Parallel algorithm for biconnecting a connected graph such that the set of edges added is a subset of a smallest triconnected augmentation.

The Complete Parallel Algorithm for G is Connected, but not Biconnected

We describe the algorithm for handling the case when the input graph is connected, but not biconnected in Algorithm 5.9. The correctness of this algorithm follows from Claim 5.5.7 and Claim 5.5.10. We know that $w(G) = O(n)$ and the number of leaves in 2-$blk(G)$ is at most n, where n is the number of vertices in the input graph. From Claim 5.5.6 and Claim 5.5.9, we know that after executing algorithm `p3aug1to2` $O(\log n)$ times, the resulting graph is biconnected.

5.5.4 The Complete Parallel Algorithm and its Implementation

In this section, we first show the complete parallel algorithm with routines for updating the 2-block graph and the 3-block graph. Then we show an efficient implementation of the parallel algorithm.

The Complete Parallel Algorithm

We present the complete parallel algorithm (Algorithm 5.10) for solving the triconnectivity augmentation problem. The correctness of algorithm `paug3` follows from the correctness we established earlier of the various cases in Claim 5.5.5, Claim 5.5.7 and Claim 5.5.10).

In the previous sections, we have shown details of each step in algo-

```
set of pairs of vertices function paug3(graph G);
{* The input graph G has at least 4 vertices; T2 is the 2-block graph;
l is the number of leaves in the 3-block graph T3. *}
   set of pairs of vertices L; forest T2, T3; set of edges S;
   Q := ∅;
   T2 := 2-blk(G); T3 := 3-blk(G);
   while T2 has more than one vertex do
      if T2 is a forest then {* G is not connected. *}
1.       L := p3aug0to1(T2)
      else if T2 is a tree and T3 is not balanced then
2.       perform a procedure similar to the one specified in Chapter 3
      else if T2 is a tree and has more than one vertex and T3 is balanced then
         {* i.e., G is connected, but not biconnected. *}
         if w(G) ≥ 56 then
3.          L := p3aug1to2(T2, T3)
         else {* w(G) < 56 *}
4.          L := ∅;
            apply the sequential triconnectivity augmentation algorithm in Section 5.4
         fi
      fi
      S := ∅; Q := Q ∪ L;
      pfor each (u, v) ∈ L do
         add an edge between u and v; S := S ∪ {(u, v)}
      rofp;
5.    par_update(T2, T3, G, S) {* The procedure par_update updates the
      2-block graph T2 and 3-block graph T3 after adding the set of edges in S. *}
   od;
   return Q ∪ paug2to3(G)
end paug3;
```

Algorithm 5.10: Complete parallel algorithm for optimally triconnecting a graph.

rithm paug3 except step 5. We now describe an algorithm for updating the 2-block graph and the 3-block graph given the current 2-block graph T_2, the current 3-block graph T_3 and the set of edges S added (step 5 in algorithm paug3).

Updating the 2-Block Graph

Note that step 1 in algorithm paug3 is executed only once and steps 2 and 4 are executed a constant number of times. Thus we can re-compute the 2-block graph and 3-block graph from the current graph G without increasing the time

complexity if algorithm `paug3` finds matched pairs by performing steps 1, 2 and 4. Note also that all 2-blocks in a fundamental cycle created by adding edges are merged into a new 2-block in the 2-block graph if algorithm `paug3` finds matched pairs by executing step 3. Hence the algorithm for updating the 2-block graph is similar to an updating algorithm given in Chapter 3.

The algorithm for updating the 3-block graph is stated in Chapter 4.

The Parallel Implementation

Let n be the number of vertices in the input graph. The 2-block graph and the 3-block graph can be constructed in $O(\log^2 n)$ time using a linear number of processors on an EREW PRAM. Note that each procedure used in algorithm `paug3` can be implemented in $O(\log n)$ time using a linear number of processors on an EREW PRAM by using the Euler technique in [TV85], procedures in [SV88] and the routine for finding triconnected components in [Ram93].

We know that steps 1, 2 and 4 are executed a constant number of times. By Claim 5.5.6 and Claim 5.5.9, we know that algorithm `paug3` biconnects the input graph by executing $O(\log n)$ times of step 3. This establishes the following claim.

Claim 5.5.11 *The triconnectivity augmentation problem on an undirected graph can be solved in time $O(\log^2 n)$ using a linear number of processors on an EREW PRAM, where n is the number vertices in the input graph.* □

5.6 Concluding Remarks

In this chapter, we have presented a linear time sequential algorithm for finding a smallest augmentation to triconnect an undirected graph. The algorithm is divided into two stages. During the first stage, we biconnect

the input graph. Then we triconnect the resulting biconnected graph using the smallest number of edges in the second stage as described in Chapter 4. We have to make sure that the total number of edges added in these two stages is minimum. Our algorithm runs in linear time. We also presented an $O(\log^2 n)$ time EREW parallel algorithm using a linear number of processors. Our parallel algorithm can be made to run within the same time bound using $O(\frac{(n+m)\log\log n}{\log n})$ processors by using the algorithm for finding connected components in [CV86] and the algorithm for integer sorting in [Hag87].

Chapter 6

Smallest Four-Connectivity Augmentation

6.1 Introduction

In this chapter, we describe a sequential algorithm for optimally four-connecting a triconnected graph. We first present a lower bound for the number of edges that must be added in order to reach four-connectivity. Note that lower bounds different from the one we give here are known for the number of edges needed to biconnect a connected graph [ET76] and to triconnect a biconnected graph (Chapter 4). It turns out that in both these cases, we can always augment the graph using exactly the number of edges specified in the lower bound. See Chapters 3 and 4 for details. However, an extension of this type of lower bound for four-connecting a triconnected graph does not always give us the exact number of edges needed [Jor92, KT91]. (For details and examples, see Section 6.3.)

We present a new type of lower bound that equals the exact number of edges needed to four-connect a triconnected graph. By using our new lower bound, we derive an $O(n\alpha(m,n)+m)$ time sequential algorithm for finding a smallest set of edges whose addition four-connects a triconnected graph with n vertices and m edges, where $\alpha(m,n)$ is the inverse Ackermann function. Our new lower bound applies for arbitrary k, and gives a tighter lower bound than the one known earlier for the number of edges needed to k-connect a $(k-1)$-connected graph. The new lower bound and the algorithm described here may lead to a better understanding of the problem of optimally k-connecting a

$(k-1)$-connected graph, for an arbitrary k. An extended abstract of the work reported here appears in [Hsu92].

6.2 Definitions

We give definitions used in this chapter.

Wheel and Flower

A set of at least three separating triplets with one common vertex c is called a *wheel* in [KTDBC91]. A wheel can be represented by the set of vertices $\{c\} \cup \{s_0, s_1, \ldots, s_{q-1}\}$ which satisfies the following conditions: (i) $q > 2$; (ii) $\forall i \neq j$, $\{c, s_i, s_j\}$ is a separating triplet unless in the case that $j = ((i+1) \bmod q)$ and (s_i, s_j) is an edge in G; (iii) c is adjacent to a vertex in each of the connected components created by removing any of the separating triplets in the wheel; (iv) $\forall j \neq (i+1) \bmod q$, $\{c, s_i, s_j\}$ is a degree-2 separating triplet. The vertex c is the *center* of the wheel [KTDBC91]. For more details, see [KTDBC91].

The *degree* of a wheel $W = \{c\} \cup \{s_0, s_1, \ldots, s_{q-1}\}$, $d(W)$, is the number of connected components in $G - \{c, s_0, \ldots, s_{q-1}\}$ plus the number of degree-3 vertices in $\{s_0, s_1, \ldots, s_{q-1}\}$ that are adjacent to c. The degree of a wheel must be at least 3. Note that the number of degree-3 vertices in $\{s_0, s_1, \ldots, s_{q-1}\}$ that are adjacent to c is equal to the number of separating triplets in $\{(c, s_i, s_{(i+2) \bmod q}) \mid 0 \leq i < q$, such that $s_{(i+1) \bmod q}$ is degree 3 in $G\}$. An example is shown in Figure 6.1.

A separating triplet is called a *flower* [KTDBC91] if it has degree > 2 or is not in any wheel. Note that it is possible that two flowers of degree-2 $f_1 = \{a_{1,i} \mid 1 \leq i \leq 3\}$ and $f_2 = \{a_{2,i} \mid 1 \leq i \leq 3\}$ have the property that $\forall i$, $1 \leq i \leq 3$, either $a_{1,i} = a_{2,i}$ or $(a_{1,i}, a_{2,i})$ is an edge in G. We write $f_1 \mathcal{R} f_2$ if f_1 and f_2 satisfy the above condition. For each flower f, the *flower cluster* $\mathcal{F}_f$ for f is the set of flowers $\{f_1, \ldots, f_x\}$ (including f) such that $f \mathcal{R} f_i$, $\forall i$, $1 \leq i \leq x$.

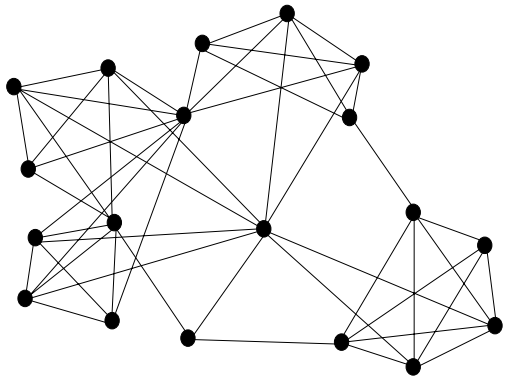


Figure 6.1: Illustrating a wheel $\{7\} \cup \{1,2,3,4,5,6\}$. The degree of this wheel is 5, i.e., the number of components we got after removing the wheel is 4 and there is one vertex (vertex 5) in the wheel with degree 3.

Each of the separating triplets in a triconnected graph G is either represented by a flower or is in a wheel. We can construct an $O(n)$-space representation for all separating triplets (i.e., flowers and wheels) in a triconnected graph with n vertices and m edges in $O(n\alpha(m,n)+m)$ time [KTDBC91].

K-Block

Let $G=(V,E)$ be a graph with vertex-connectivity $k-1$. A *k-block* in G is either (i) a minimal set of vertices $\mathcal{B}$ in a separating $(k-1)$-set with exactly $k-1$ neighbors in $V \setminus \mathcal{B}$ (these are *special k-blocks*) or (ii) a maximal set of vertices $\mathcal{B}$ such that there are at least k vertex-disjoint paths in G between any two vertices in $\mathcal{B}$ and $\mathcal{B}$ is not a special k-block (these are *non-special k-blocks*). Note that a set consisting of a single vertex of degree $k-1$ in G is a k-block. A *k-block leaf* in G is a k-block $\mathcal{B}_l$ with exactly $k-1$ neighbors in $V \setminus \mathcal{B}_l$. Note also that every special k-block is a k-block leaf. If there is any special 4-block in a separating triplet $\mathcal{S}$, $d(\mathcal{S}) \leq 3$. Given a non-special 4-block $\mathcal{B}$ leaf, the vertices in $\mathcal{B}$ that are not in the flower cluster that separates $\mathcal{B}$ are *demanding vertices*. We let every vertex in a special 4-block leaf be a demanding vertex.

Claim 6.2.1 *Every non-special 4-block leaf contains at least one demanding vertex.* □

Using procedures in [KTDBC91], we can find all of the 4-block leaves in a triconnected graph with n vertices and m edges in $O(n\alpha(m,n)+m)$ time.

Four-Block Tree

From [KTDBC91] we know that we can decompose vertices in a triconnected graph into the following 3 types: (i) 4-blocks; (ii) wheels; (iii) separating triplets that are not in a wheel. We modify the decomposition tree in [KTDBC91] to derive the *four-block tree* 4-$blk(G)$ for a triconnected graph G as follows. We create an R-vertex for each 4-block that is not special (i.e., not in a separating set or in the center of a wheel), an F-vertex for each separating triplet that is not in a wheel, and a W-vertex for each wheel. For each wheel $W = \{c\} \cup \{s_0, s_1, \ldots, s_{q-1}\}$, we also create the following vertices. An F-vertex is created for each separating triplet of the form $\{c, s_i, s_{(i+1) \bmod q}\}$ in W. An R-vertex is created for every degree-3 vertex s in $\{s_0, s_1, \ldots, s_{q-1}\}$ that is adjacent to c and an F-vertex is created for the three vertices that are adjacent to s. There is an edge between an F-vertex f and an R-vertex r if each vertex in the separating triplet corresponding to f is either in the 4-block H_r corresponding to r or adjacent to a vertex in H_r. There is an edge between an F-vertex f and a W-vertex w if the the wheel corresponding to w contains the separating triplet corresponding to f. A *dummy R-vertex* is created and adjacent to each pair of flowers f_1 and f_2 with the properties that f_1 and f_2 are not already connected and their flower clusters contain each other (i.e., $f_1 \in \mathcal{F}_{f_2}$ and $f_2 \in \mathcal{F}_{f_1}$). An example of a 4-block tree is shown in Figure 6.2.

Note that a degree-1 R-vertex in 4-$blk(G)$ corresponds to a 4-block leaf, but the reverse is not necessarily true, since we do not represent some

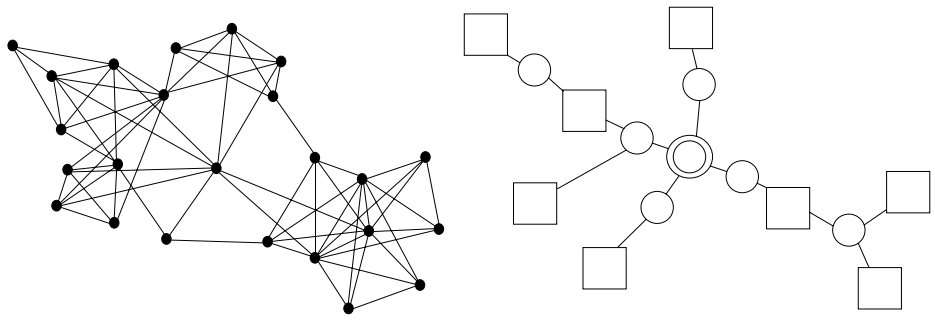


Figure 6.2: Illustrating a triconnected graph and its 4-$blk(G)$. We use rectangles, circles and two concentric circles to represent R-vertices, F-vertices and W-vertices, respectively. The vertex-numbers beside each vertex in 4-$blk(G)$ represent the set of vertices corresponding to this vertex.

special 4-block leaves and all degree-3 vertices that are centers of wheels in 4-$blk(G)$. A special 4-block leaf $\{v\}$, where v is a vertex, is represented by an R-vertex in 4-$blk(G)$ if v is not the center of a wheel w and it is in one of separating triplets of w. The degree of a flower F in G is the degree of its corresponding vertex in 4-$blk(G)$. Note also that the degree of a wheel W in G is equal to the number of components in 4-$blk(G)$ by removing its corresponding W-vertex w and all F-vertices that are adjacent to w. A wheel W in G is a *star wheel* if $d(W)$ equals the number of leaves in 4-$blk(G)$ and every special 4-block leaf in W is either adjacent to or equal to the center. A star wheel W with the center c has the property that every 4-block leaf in G (not including $\{c\}$ if it is a 4-block leaf) can be separated from G by a separating triplet containing the center c. If G contains a star wheel W, then W is the only wheel in G. Note also that the degree of a wheel is less than or equal to the degree of its center in G.

K-Connectivity Augmentation Number

The *k-connectivity augmentation number* for a graph G is the smallest number

of edges that must be added to G in order to k-connect G.

6.3 A Lower Bound for the Four-Connectivity Augmentation Number

We first give a simple lower bound for the four-connectivity augmentation number that is similar to the ones for biconnectivity augmentation [ET76] and triconnectivity augmentation as described in Chapter 4. We show that this above lower bound is not always equal to the four-connectivity augmentation number [Jor92, KT91]. We then give a modified lower bound. This new lower bound turns out to be the exact number of edges that we must add to reach four-connectivity (see proofs in Section 6.4). Finally, we show relations between the two lower bounds.

6.3.1 A Simple Lower Bound

Given a graph G with vertex-connectivity $k-1$, it is well-known that $\max\{\lceil \frac{l_k}{2} \rceil, d-1\}$ is a lower bound for the k-connectivity augmentation number where l_k is the number of k-block leaves in G and d is the maximum degree among all separating $(k-1)$-sets in G [ET76]. It is also well-known (see, for example, Chapters 3 and 4) that for $k = 2$ and 3, this lower bound equals the k-connectivity augmentation number. For $k = 4$, however, several researchers [Jor92, KT91] have observed that this value is not always equal to the four-connectivity augmentation number. Examples are given in Figure 6.3. Figure 6.3.(1) is from [Jor92] and Figure 6.3.(2) is from [KT91]. Note that if we apply the above lower bound in each of the three graphs in Figure 6.3, the values we obtain for Figures 6.3.(1), 6.3.(2), and 6.3.(3) are 3, 3, and 2, respectively, while we need one more edge in each graph to four-connect it.

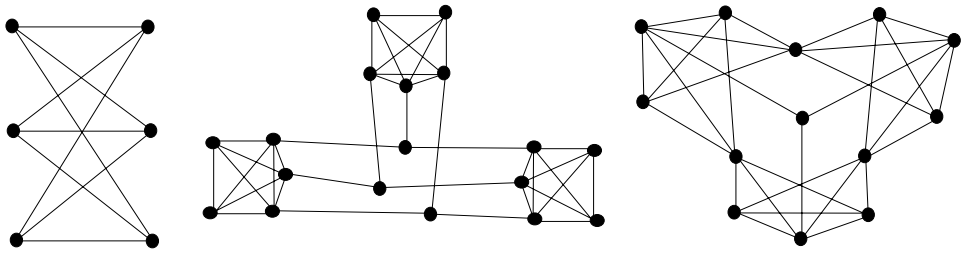


Figure 6.3: Illustrating three graphs where in each case the value derived by applying a simple lower bound does not equal its four-connectivity augmentation number.

6.3.2 A Better Lower Bound

Notice that in the previous lower bound, for every separating triplet $\mathcal{S}$ in the triconnected graph $G = \{V, E\}$, we must add at least $d(\mathcal{S}) - 1$ edges between vertices in $V \setminus \mathcal{S}$ to four-connect G, where $d(\mathcal{S})$ is the degree of $\mathcal{S}$ (i.e., the number of connected components in $G - \mathcal{S}$); otherwise, $\mathcal{S}$ remains a separating triplet. Let the set of edges added be $\mathcal{A}_{1,\mathcal{S}}$. We also notice that we must add at least one edge into every 4-block leaf $\mathcal{B}$ to four-connect G; otherwise, $\mathcal{B}$ remains a 4-block leaf. Since it is possible that $\mathcal{S}$ contains some 4-block leaves, we need to know the minimum number of edges needed to eliminate all 4-block leaves inside $\mathcal{S}$. Let the set of edges added be $\mathcal{A}_{2,\mathcal{S}}$. We know that $\mathcal{A}_{1,\mathcal{S}} \cap \mathcal{A}_{2,\mathcal{S}} = \emptyset$. The previous lower bound gives a bound on the cardinality of $\mathcal{A}_{1,\mathcal{S}}$, but not that of $\mathcal{A}_{2,\mathcal{S}}$. In the following paragraph, we define a quantity to measure the cardinality of $\mathcal{A}_{2,\mathcal{S}}$.

Let $\mathcal{Q}_{\mathcal{S}}$ be the set of special 4-block leaves that are in the separating triplet $\mathcal{S}$ of a triconnected graph G. Two 4-block leaves $\mathcal{B}_1$ and $\mathcal{B}_2$ are *adjacent* if there is an edge in G between every demanding vertex in $\mathcal{B}_1$ and every demanding vertex in $\mathcal{B}_2$. We create an *augmenting graph for* $\mathcal{S}$, $\mathcal{G}(\mathcal{S})$, as follows. For each special 4-block leaf in $\mathcal{Q}_{\mathcal{S}}$, we create a vertex in $\mathcal{G}(\mathcal{S})$. There

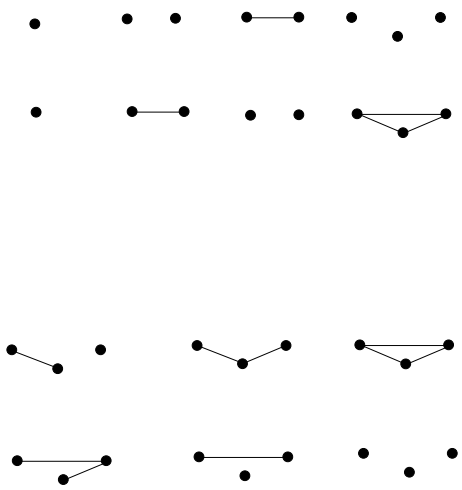


Figure 6.4: Illustrating the seven types of augmenting graphs, their complement graphs and augmenting numbers that one can get for a separating triplet in a triconnected graph.

is an edge between two vertices v_1 and v_2 in $\mathcal{G}(\mathcal{S})$ if their corresponding 4-blocks are adjacent. Let $\overline{\mathcal{G}(\mathcal{S})}$ be the complement graph of $\mathcal{G}(\mathcal{S})$. The seven types of augmenting graphs and their complement graphs are illustrated in Figure 6.4.

Definition 6.3.1 *The* **augmenting number** $a(\mathcal{S})$ *for a separating triplet* $\mathcal{S}$ *in a triconnected graph is the number of edges in a maximum matching* $\mathcal{M}$ *of* $\overline{\mathcal{G}(\mathcal{S})}$ *plus the number of vertices that have no edges in* $\mathcal{M}$ *incident on them.*

The augmenting numbers for the seven types of augmenting graphs are shown in Figure 6.4. Note that in a triconnected graph, each special 4-block leaf must receive at least one new incoming edge in order to four-connect the input graph. The augmenting number $a(\mathcal{S})$ is exactly the minimum number of edges needed in the separating triplet $\mathcal{S}$ in order to four-connect the input graph.

The augmenting number of a separating set that does not contain any special 4-block leaf is 0. Note also that we can define the *augmenting number* $a(\mathcal{C})$ for a set $\mathcal{C}$ that consists of the center of a wheel using a similar approach. Note that $a(\mathcal{C}) \leq 1$.

We also need the following definition.

Definition 6.3.2 *Let G be a triconnected graph with l 4-block leaves. The* **leaf constraint** *of G, $lc(G)$, is $\lceil \frac{l}{2} \rceil$. The* **degree constraint** *of a separating triplet S in G, $dc(\mathcal{S})$, is $d(\mathcal{S})-1+a(\mathcal{S})$, where $d(\mathcal{S})$ is the degree of $\mathcal{S}$ and $a(\mathcal{S})$ is the augmenting number of $\mathcal{S}$. The degree constraint of G, $dc(G)$, is the maximum degree constraint among all separating triplets in G. The* **wheel constraint** *of a star wheel W with center c in G, $wc(W)$, is $\lceil \frac{d(W)}{2} \rceil + a(\{c\})$, where $d(W)$ is the degree of W and $a(\{c\})$ is the augmenting number of $\{c\}$. The wheel constraint of G, $wc(G)$, is 0 if there is no star wheel in G; otherwise it is the wheel constraint of the star wheel in G.*

We now give a better lower bound on the 4-connectivity augmentation number for a triconnected graph.

Lemma 6.3.3 *We need at least* $\max\{lc(G), dc(G), wc(G)\}$ *edges to four-connect a triconnected graph G.*

Proof. Let $\mathcal{A}$ be a set of edges such that $G' = G \cup \mathcal{A}$ is four-connected. For each 4-block leaf B in G, we need one new incoming edge to a vertex in B; otherwise B is still a 4-block leaf in G'. This gives the first component of the lower bound.

For each separating triplet $\mathcal{S}$ in G, $G - \mathcal{S}$ contains $d(\mathcal{S})$ connected components. We need to add at least $d(\mathcal{S})-1$ edges between vertices in $G-\mathcal{S}$,

otherwise $\mathcal{S}$ is still a separating triplet in G'. In addition to that, we need to add at least $a(\mathcal{S})$ edges such that at least one of the two end points of each new edge is in $\mathcal{S}$; otherwise $\mathcal{S}$ contains a special 4-block leaf. This gives the second term of the lower bound.

Given the star wheel W with the center c, 4-$blk(G)$ contains exactly $d(W)$ degree-1 R-vertices. Thus we need to add at least $\lceil \frac{d(W)}{2} \rceil$ edges between vertices in $G - \{c\}$; otherwise, G' contains some 4-block leaves. In addition to that, we need to add $a(\{c\})$ non-self-loop edges such that at least one of the two end points of each new edge is in $\{c\}$; otherwise $\{c\}$ is still a special 4-block leaf. This gives the third term of the lower bound. □

6.3.3 A Comparison of the Two Lower Bounds

We first observe the following relation between the wheel constraint and the leaf constraint. Note that if there exists a star wheel W with degree $d(W)$, there are exactly $d(W)$ 4-block leaves in G if the center is not degree-3. If the center of the star wheel is degree-3, then there are exactly $d(W) + 1$ 4-block leaves in G. Thus the wheel constraint is greater than the leaf constraint if and only if the star wheel has a degree-3 center. We know that the degree of any wheel is less than or equal to the degree of its center. Thus the value of the above lower bound equals 3.

We state the following claims for the relations between the degree constraint of a separating triplet and the leaf constraint.

Claim 6.3.4 *Let $\mathcal{S}$ be a separating triplet with degree $d(\mathcal{S})$ and h special 4-block leaves. Then there are at least $h + d(\mathcal{S})$ 4-block leaves in G.* □

Claim 6.3.5 *Let $\{a_1, a_2, a_3\}$ be a separating triplet in a triconnected graph G. Then a_i, $1 \leq i \leq 3$, is incident on a vertex in every connected component in $G - \{a_1, a_2, a_3\}$.* □

Corollary 6.3.6 *The degree of a separating triplet $\mathcal{S}$ is no more than the largest degree among all vertices in $\mathcal{S}$.* □

From Corollary 6.3.6, we know that it is not possible that a triconnected graph has type (6) or type (7) of the augmenting graphs as shown in Figure 6.4, since the degree of their underling separating triplet is 1. We also know that the degree of a separating triplet with a special 4-block leaf is at most 3 and at least 2. Thus $dc(\mathcal{S})$ is greater than $d(\mathcal{S}) - 1$ if $dc(\mathcal{S})$ equals either 3 or 4. Thus we have the following lemma.

Lemma 6.3.7 *Let $low_1(G)$ be the lower bound given in Section 6.3.1 for a triconnected graph G and let $low_2(G)$ be the lower bound given in Lemma 6.3.3 in Section 6.3.2. (i) $low_1(G) = low_2(G)$ if $low_2(G) \notin \{3, 4\}$. (ii) $low_2(G) - low_1(G) \in \{0, 1\}$.* □

Thus the simple lower bound extended from biconnectivity and triconnectivity is in fact a good approximation for the four-connectivity augmentation number.

6.4 Finding a Smallest Four-Connectivity Augmentation for a Triconnected Graph

We first explore properties of the 4-block tree that we will use in this section to develop an algorithm for finding a smallest 4-connectivity augmentation. Then we describe our algorithm. Graphs discussed in this section are triconnected unless specified otherwise.

6.4.1 Properties of the Four-Block Tree

Massive Vertex, Critical Vertex and Balanced Graph

A separating triplet $\mathcal{S}$ in a graph G is *massive* if $dc(\mathcal{S}) > lc(G)$. A separating triplet $\mathcal{S}$ in a graph G is *critical* if $dc(\mathcal{S}) = lc(G)$. A graph G is *balanced* if there is no massive separating triplet in G. If G is balanced, then its 4-$blk(G)$ is also *balanced*. The following lemma and corollary state the number of massive and critical vertices in 4-$blk(G)$.

Lemma 6.4.1 *Let $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ be any three separating triplets in G such that there is no special 4-block in $\mathcal{S}_i \cap \mathcal{S}_j$, $1 \leq i < j \leq 3$. $\sum_{i=1}^{3} dc(\mathcal{S}_i) \leq l + 1$, where l is the number of 4-block leaves in G.*

Proof. The input graph G is triconnected. We can modify 4-$blk(G)$ in the following way such that the number of leaves in the resulting tree equals l and the degree of an F-node f equals its degree constraint plus 1 if f corresponds to $\mathcal{S}_i$, $1 \leq i \leq 3$. For each W-vertex w with a degree-3 center c, we create an R-vertex r_c for c, an F-vertex f_c for the three vertices that are adjacent to c in G. We add edges (w, f_c) and (f_c, r_c). Thus r_c is a leaf. For each F-vertex whose corresponding separating triplet $\mathcal{S}$ contains h special 4-block leaves, we attach $a(\mathcal{S})$ subtrees with a total number of h leaves with the constraint that any special 4-block that is in more than one separating triplet will be added only once (to the F-node corresponding to $\mathcal{S}_i$, $1 \leq i \leq 3$, if possible). From Figure 6.4 we know that the number of special 4-block leaves in any separating triplet is greater than or equal to its augmenting number. Thus the above addition of subtrees can be done. Let 4-$blk(G)'$ be the resulting graph. Thus the number of leaves in 4-$blk(G)'$ is l. Let f be an F-node in 4-$blk(G)'$ whose corresponding separating triplet is $\mathcal{S}$. We know that the degree of f equals

$dc(\mathcal{S})+1$ if $\mathcal{S} \in \{\mathcal{S}_i \mid 1 \leq i \leq 3\}$. It is easy to verify that the sum of degrees of any three internal vertices in a tree is less than or equal to 4 plus the number of leaves in a tree. □

Corollary 6.4.2 *Let G be a graph with more than two non-special 4-block leaves. (i) There is at most one massive F-vertex in 4-blk(G). (ii) If there is a massive F-vertex, there is no critical F-vertex. (iii) There are at most two critical F-vertices in 4-blk(G).* □

Updating the Four-Block Tree

Let v_i be a demanding vertex or a vertex in a special 4-block leaf, $i \in \{1,2\}$. Let $\mathcal{B}_i$ be the 4-block leaf that contains v_i, $i \in \{1,2\}$. Let b_i, $i \in \{1,2\}$, be the vertex in 4-$blk(G)$ such that if v_i is a demanding vertex, then b_i is an R-vertex whose corresponding 4-block contains v_i; if v_i is in a special 4-block leaf in a flower, then b_i is the F-vertex whose corresponding separating triplet contains v_i; if v_i is the center of a wheel w, b_i is the F-vertex that is closet to $b_{(i \bmod 2)+1}$ and is adjacent to w. The vertex b_i is the *implied vertex* for $\mathcal{B}_i$, $i \in \{1,2\}$. The *implied path P between $\mathcal{B}_1$ and $\mathcal{B}_2$* is the path in 4-$blk(G)$ between b_1 and b_2. Given 4-$blk(G)$ and an edge (v_1, v_2) not in G, we can obtain 4-$blk(G \cup \{(v_1, v_2)\})$ by performing local updating operations on P. For details, see [KTDBC91].

In summary, all 4-blocks corresponding to R-vertices in P are collapsed into a single 4-block. Edges in P are deleted. F-vertices in P are connected to the new R-vertex created. We *crack* wheels in a way that is similar to the cracking of a polygon for updating 3-block graphs (see Chapter 4 and [DBT90] for details). We say that P is *non-adjacent* on a wheel W, if the cracking of W creates two new wheels. Note that it is possible that a separating triplet $\mathcal{S}$ in the original graph is no longer a separating triplet in the resulting

graph by adding an edge. Thus some special leaves in the original graph are no longer special, in which case they must be added to 4-$blk(G)$.

Reducing the Degree Constraint of a Separating Triplet

We know that the degree constraint of a separating triplet can be reduced by at most 1 by adding a new edge. From results in [KTDBC91], we know that we can reduce the degree constraint of a separating triplet $\mathcal{S}$ by adding an edge between two non-special 4-block leaves $\mathcal{B}_1$ and $\mathcal{B}_2$ such that the path in 4-$blk(G)$ between the two vertices corresponding to $\mathcal{B}_1$ and $\mathcal{B}_2$ passes through the vertex corresponding to $\mathcal{S}$. We also notice the following corollary from the definitions of 4-$blk(G)$ and the degree constraint.

Corollary 6.4.3 *Let $\mathcal{S}$ be a separating triplet that contains a special 4-block leaf. (i) We can reduce $dc(\mathcal{S})$ by 1 by adding an edge between two special 4-block leaves $\mathcal{B}_1$ and $\mathcal{B}_2$ in $\mathcal{S}$ such that $\mathcal{B}_1$ and $\mathcal{B}_2$ are not adjacent. (ii) If we add an edge between a special 4-block leaf in $\mathcal{S}$ and a 4-block leaf $\mathcal{B}$ not in $\mathcal{S}$, the degree constraint of every separating triplet corresponding to an internal vertex in the path of 4-$blk(G)$ between vertices corresponding to $\mathcal{S}$ and $\mathcal{B}$ is reduced by 1.* □

Note that part (i) in Corollary 6.4.3 can be verified by observing all different augmenting graphs for a triconnected graph (shown in Figure 6.4).

Reducing the Number of Four-Block Leaves

We now consider the conditions under which the adding of an edge reduces the leaf constraint $lc(G)$ by 1. Let *real degree* of an F-node in 4-$blk(G)$ be 1 plus the degree constraint of its corresponding separating triplet. The real degree of a W-node with a degree-3 center in G is 1 plus its degree in 4-$blk(G)$. The real degree of any other node is equal to its degree in 4-$blk(G)$.

Definition 6.4.4 (The leaf-connecting condition) *Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two non-adjacent 4-block leaves in G. Let P be the implied path between $\mathcal{B}_1$ and $\mathcal{B}_2$ in 4-blk(G). Two 4-block leaves $\mathcal{B}_1$ and $\mathcal{B}_2$ satisfy the* **leaf-connecting condition** *if at least one of the following conditions is true. (i) There are at least two vertices of real degree at least 3 in P. (ii) There is at least one R-vertex of degree at least 4 in P. (iii) The path P is non-adjacent on a W-vertex in P. (iv) There is an internal vertex of real degree at least 3 in P and at least one of the 4-block leaves in $\{\mathcal{B}_1, \mathcal{B}_2\}$ is special. (v) $\mathcal{B}_1$ and $\mathcal{B}_2$ are both special and they do not share the same set of neighbors.*

Lemma 6.4.5 *Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two 4-block leaves in G that satisfy the leaf-connecting condition. We can find vertices v_i in $\mathcal{B}_i$, $i \in \{1,2\}$, such that $lc(G \cup \{(v_1, v_2)\}) = lc(G) - 1$, if $lc(G) \geq 2$.*

Proof. Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be the two 4-block leaves that satisfy the leaf-connecting condition. If they satisfy parts i to iii of the leaf-connecting, proofs similar to the ones given in Chapter 4 for finding a smallest triconnectivity augmentation can be used to prove this lemma.

Assume that $\mathcal{B}_1$ and $\mathcal{B}_2$ satisfy part (iv) or part (v) of the leaf-connecting condition. Since we add an edge between $\mathcal{B}_1$ and $\mathcal{B}_2$, $\mathcal{B}_i$, $i \in \{1,2\}$, is no longer a 4-block leaf. We have to show that the new 4-block created is not a 4-block leaf. If $\mathcal{B}_1$ and $\mathcal{B}_2$ satisfy part (iv) of the leaf-connecting condition, then the new 4-block created is adjacent to at least two F-vertices. One of them is the degree-3 vertex q in P if q is an F-vertex; otherwise it is an F-node adjacent to q. The other F-vertex adjacent to the created 4-block is the F-vertex contains the special 4-block leaf in P.

If $\mathcal{B}_1$ and $\mathcal{B}_2$ satisfy part (v) of the leaf-connecting condition and they are in the same separating triplet, then no 4-block is created. Otherwise, the

created 4-block is adjacent to the two F-vertices whose corresponding separating triplets contain $\mathcal{B}_1$ and $\mathcal{B}_2$. □

6.4.2 The Algorithm

We now describe an algorithm for finding a smallest augmentation to four-connect a triconnected graph. Let $\delta = dc(G) - lc(G)$. The algorithm first adds 2δ edges to the graph such that the resulting graph is balanced and the lower bound is reduced by 2δ. If $lc(G) \neq 2$ or $wc(G) \neq 3$, there is no star wheel with a degree-3 center. We add an edge such that the degree constraint $dc(G)$ is reduced by 1 and the number of 4-block leaves is reduced by 2. Since there is no star wheel with a degree-3 center, $wc(G)$ is also reduced by 1 if $wc(G) = lc(G)$. The resulting graph stays balanced each time we add an edge and the lower bound given in Lemma 6.3.3 is reduced by 1. If $lc(G) = 2$ and $wc(G) = 3$, then there exists a star wheel with a degree-3 center. We reduce $wc(G)$ by 1 by adding an edge between the degree-3 center and a demanding vertex of a 4-block leaf. Since $lc(G) = 2$ and $wc(G) = 3$, $dc(G)$ is at most 2. Thus the lower bound can be reduced by 1 by adding an edge. We keep adding an edge at a time such that the lower bound given in Lemma 6.3.3 is reduced by 1. Thus we can find a smallest augmentation to four-connect a triconnected graph. We now describe our algorithm.

Input Graph is Not Balanced

We use an approach that is similar to the one used in biconnectivity (Chapter 3) and triconnectivity augmentations (Chapter 4) to balance the input graph. Given a tree T and a vertex v in T, a *v-chain* [RG77] is a component in $T - \{v\}$ without any vertex of degree more than 2. The leaf of T in each v-chain is a *v-chain leaf* [RG77]. Let $\delta = dc(G) - lc(G)$ for a unbalanced graph G and let

4-$blk(G)'$ be the modified 4-block tree given in the proof of Lemma 6.4.1. Let f be a massive F-vertex. We can show that either there are at least $2\delta + 2$ f-chains in 4-$blk(G)'$ (i.e., f is the only massive F-vertex) or we can eliminate all massive F-vertices by adding an edge. Let λ_i be a demanding vertex in the ith f-chain leaf. We add the set of edges $\{(\lambda_i, \lambda_{i+1}) \mid 1 \leq i \leq 2\delta\}$. It is also easy to show that the lower bound given in Lemma 6.3.3 is reduced by 2δ and the graph is balanced.

Input Graph is Balanced

We first describe the algorithm in Algorithm 6.1. Note that Algorithm 6.1 uses a subroutine shown in Algorithm 6.2 to handle the case that the 4-block graph is a star. Then we give its proof of correctness. In the description, we need the following definition. Let $\mathcal{B}$ be a 4-block leaf whose implied vertex in 4-$blk(G)$ is b and let $\mathcal{B}'$ be a 4-block leaf whose implied vertex in 4-$blk(G)$ is b'. The leaf $\mathcal{B}'$ is a *nearest* 4-block leaf of $\mathcal{B}$ if there is no other 4-block leaf whose implied vertex has a distance to b that is shorter than the distance between b and b'.

Before we show the correctness of algorithm `aug3to4`, we need the following claim and corollaries.

Claim 6.4.6 *[RG77] If 4-$blk(G)$ contains two critical vertices f_1 and f_2, then every leaf is either in an f_1-chain or in an f_2-chain and the degree of any other vertex in 4-$blk(G)$ is at most 2.* □

Corollary 6.4.7 *Let f_1 and f_2 be two critical vertices in 4-$blk(G)$ and let $\mathcal{S}_i$, $i \in \{1,2\}$, be the corresponding separating triplet of f_i. If $\mathcal{S}_i$, $i \in \{1,2\}$, f_i contains a special 4-block leaf, then the augmenting number of f_i is equal to the number of special 4-block leaves in $\mathcal{S}_i$.*

```
graph function aug3to4(graph G);
   T := 4-blk(G); root T at an arbitrary vertex; l̃ := |{ degree-1 R-vertices in T }|;
   while ∃ a 4-block leaf in G do
      if ∃ a degree-3 center c then
1.       if lc(G) = 2 and wc(G) = 3 then {* c is the center of the star wheel w. *}
            u1 := the 4-block leaf {c}; let u2 be a a non-special 4-block leaf
         else if ∃ another degree-3 center c' non-adjacent to c then
            let u2 be the 4-block leaf {c'}
         else if ∃ a special 4-block leaf b non-adjacent to u1 then let u2 := b
         else if ∄ (degree-3 center or special 4-block leaf) non-adjacent to u1 then
            let u2 be a a 4-block leaf s. t. ∃ an internal vertex with
            real degree ≥ 3 in their implies path fi
      else if lc(G) ≠ 2 or wc(G) ≠ 3 then
         if l̃ > 2 and ∃ 2 critical F-vertices f1 and f2 then
2.          find two non-special 4-block leaves u1 and u2 s. t. the implied path
            between them passes through f1 and f2
         else if l > 2 and ∃ only one critical F-vertex f1 then
            if ∃ two non-adjacent special 4-block leaves in the
            separating triplet S1 corresponding to f1 then
3.             let u1 and u2 be two non-adjacent 4-block leaves in S1
            else if ∄ two non-adjacent special 4-block leaves in the
            separating triplet S1 corresponding to f1 then
4.             let v be a vertex with the largest real degree among all vertices in T − f1;
               if real degree of v in T ≥ 3 then
                  find two non-special 4-block leaves u1 and u2
                  s. t. the implied path between them passes through f1 and v fi
            fi{* The case when the degree of v in T < 3 will be handled in step 8. *}
         else if ∃ two vertices v1 and v2 with real degree ≥ 3 then
5.          find two non-special 4-block leaves u1 and u2 such
            that the implied path between them passes through v1 and v2
         else if ∃ an R-vertex v of degree ≥ 4 then
6.          find two non-special 4-block leaves u1 and u2 such
            that the implied path between them passes through v
         else if ∃ a W-vertex v of degree ≥ 4 then
7.          let u1 and u2 be two non-special 4-block leaves such
            that the implied path between them is non-adjacent on v
         else {* T is a star with the center v. *} star(u1, u2, l̃, T)
      fi;
      let yi, i ∈ {1,2}, be a demanding vertex in ui s. t.
      (y1, y2) is not an edge in the current G;
      G := G ∪ {(y1, y2)}; update T, l̃, lc(G), wc(G) and dc(G)
   od;
   return G
end aug3to4;
```

Algorithm 6.1: Algorithm for finding a smallest four-connectivity augmentation of a triconnected graph.

```
{* The input 4-block tree T is a star. Find u1 and u2 in T
such that we can connect them and reduce the augmentation number. *}
procedure star(modifies vertex u1, u2, integer l̃, tree T);
        if there is one vertex v in T with degree ≥ 3 then
8.        find a nearest vertex w of v that contains a 4-block leaf v1;
          let w' be a nearest vertex of w containing a 4-block leaf non-adjacent to v1;
          find 4-block leaves u1 and u2 whose implied path passes through w, w' and v
          {* The above step can be always done, since T is a star. *}
        {* Note that T is path for all the cases below. *}
        else if ∃ 2 non-adjacent special 4-block leaves in a separating triplet S then
9.        let u1 and u2 be two non-adjacent special 4-block leaves in S
        else if ∃ a special 4-block leaf u1 then
10.       find a nearest non-adjacent 4-block leaf u2
        else {* l̃ = 2 *}
          let u1 and u2 be the two 4-block leaves
          corresponding to the two degree-1 R-vertices in T
        fi
end star;
```

Algorithm 6.2: A subroutine called by algorithm `aug3to4` to handle the case when the 4-block graph is a star.

Proof. It is easy to check that Claim 6.4.6 is true for the modified 4-block tree we gave in the proof of Lemma 6.4.1. We observe from Figure 6.4 that the augmenting number of a separating triplet is at most equal to the number of special 4-block leaves in it. If we have more special 4-block leaves than its augmenting number, then the modified 4-block tree we built does not satisfy the condition imposed by Claim 6.4.6. □

Corollary 6.4.8 *Let f_1 and f_2 be two critical F-vertices in 4-blk(G). If the number of degree-1 R-vertices in 4-blk(G) > 2 and the corresponding separating triplet of f_i, $i \in \{1,2\}$, contains a 4-block leaf $\mathcal{B}_i$, we can add an edge between a vertex in $\mathcal{B}_1$ and a vertex in $\mathcal{B}_2$ to reduce the lower bound given in Lemma 6.3.3 by 1.* □

Theorem 6.4.9 *Algorithm* `aug3to4` *adds the smallest number of edges to four-connect a triconnected graph.*

Proof. We first observe that if the wheel constraint $wc(G)$ dominates the lower bound, then there exists exactly one wheel w. The wheel w is a star wheel and has a degree-3 center. We also know that 4-$blk(G)$ contains 3 non-special 4-block leaves and there is no critical F-vertex. The pair of vertices found in step 1 satisfy part (iv) or part (v) of the leaf-connecting condition. Thus step 1 of algorithm `aug3to4` finds the right pair of vertices between which a new edge is added if $wc(G)$ dominates.

If the degree constraint dominates, then there is at least one critical vertex. Steps 2, 3, 4, 8 and 9 make sure the degree constraint of any critical vertex is reduced by 1 by adding the new edge found. (Note that steps 8, 9 and 10 are in Algorithm 6.2.) Corollary 6.4.8 makes sure the implied path between the pair of vertices found in step 9 passes through all critical vertices, if any. The pair of vertices found in steps 2 and 4 satisfy part (i) of the leaf-connecting condition. The pair of vertices found in steps 3 and 8 satisfy part (iv) of the leaf-connecting condition. The pair of vertices found in step 9 satisfy part (v) of the leaf-connecting satisfy part (iv) of the leaf-connecting condition. The pair of vertices found in step 9 satisfy part (v) of the leaf-connecting condition. Thus we reduce both $dc(G)$ and $lc(G)$ by 1. Hence the lower bound is reduced by 1 by adding an edge.

We now prove the case when the leaf constraint dominates. We have to make sure the pair of vertices found satisfy the leaf-connecting condition. In the following, we show in each step, the part of the leaf-connecting condition that is satisfied if the number of 4-block leaves is at least 4. Step 2: part (i); step 3: part (v); step 4: part (i); step 5: part (i); step 6: part (ii); step 7: part (iii); step 8: part (iv) or part (v); step 9: part (v); step 10: part (v). If there

are less than 3 4-block leaves in G, we can add an edge between demanding vertices of any arbitrary two 4-block leaves. Thus $lc(G)$ is reduced by 1 each time we add an edge. Hence the lower bound is reduced by 1 by adding an edge. □

We now describe an efficient way of implementing algorithm `aug3to4`. The 4-block tree can be computed in $O(n\alpha(m,n)+m)$ time for a graph with n vertices and m edges [KTDBC91]. We know that the leaf constraint, the degree constraint of any separating triplet and the wheel constraint of any wheel in G can only be decreased by adding an edge. We also know that $lc(G)$, the sum of degree constraints of all separating triplets and the sum of wheel constraints of all wheels are all $O(n)$. Thus we can use the technique in [RG77] to maintain the current leaf constraint, the degree constraint for any separating triplet and the wheel constraint for any wheel in $O(n)$ time for the entire execution of the algorithm. We also visit each vertex and each edge in the 4-block tree a constant number of times before deciding to collapse them. There are $O(n)$ 4-block leaves and $O(n)$ vertices and edges in 4-$blk(G)$. We use a set-union-find algorithm to maintain the identities of vertices after collapsing. Hence the overall time for updating the 4-block tree is $O(n\alpha(n,n))$. We have the following claim.

Claim 6.4.10 *Algorithm* `aug3to4` *can be implemented in* $O(n\alpha(m,n)+m)$ *time where* n *and* m *are the number of vertices and edges in the input graph, respectively and* $\alpha(m,n)$ *is the inverse Ackermann function.* □

6.5 Concluding Remarks

We have given a sequential algorithm for finding a smallest set of edges whose addition four-connects a triconnected graph. The algorithm runs

in $O(n\alpha(m,n)+m)$ time using $O(n+m)$ space. The following approach was used in developing our algorithm. We first gave a 4-block tree data structure for a triconnected graph that is similar to the one given in [KTDBC91]. We then described a lower bound on the smallest number of edges that must be added based on the 4-block tree of the input graph. We further showed that it is possible to decrease this lower bound by 1 by adding an appropriate edge.

The lower bound that we gave here is different from the ones that we have for biconnecting a connected graph and for triconnecting a biconnected graph. We also showed relations between these two lower bounds. This new lower bound applies for arbitrary k, and gives a tighter lower bound than the one known earlier for the number of edges needed to k-connect a $(k-1)$-connected graph. It is likely that techniques presented in this chapter may be used in finding the k-connectivity augmentation number of a $(k-1)$-connected graph, for an arbitrary k.

Chapter 7

Smallest Edge-Connectivity Augmentation

7.1 Introduction

In this chapter, we consider algorithms for finding smallest edge-connectivity augmentations. We present a linear time algorithm and an efficient parallel algorithm to construct a compact representation for all separating edge-pairs in an undirected graph G that is 2-edge-connected. Our parallel algorithm runs in $O(\log n)$ time on a CRCW PRAM using $O(\frac{(n+m)\log\log n}{\log n})$ processors on a CRCW PRAM, where n and m are the number of vertices and edges in G, respectively. We then use this above algorithm together with results in Chapters 4 and 5, and [ET76, NGM90, Ram93] to derive an algorithm with the same complexity bounds to find a smallest set of edges whose addition 3-edge-connects a graph. We will also show a simple linear time algorithm to construct a compact representation for all separating edge-triplets in a graph G that is not 4-edge-connected given compact representations for all cutpoints, separating vertex-pairs, and separating vertex-triplets in G. By using our result and results in [NGM90], we can find a smallest set of edges whose addition 4-edge-connects a graph in the same complexity as the algorithm for building a compact representation for all separating vertex-triplets [KTDBC91] (which runs in in $O(n\alpha(m,n)+m)$ time sequentially). We do not have an efficient parallel algorithm for this problem since there is no efficient parallel algorithm known for finding a compact representation for all separating vertex-triplets in a graph.

Previous results for solving the above problems are as follows. Gabow [Gab91] gave an $O(m + n \log n)$ time sequential algorithm for constructing a structure similar to ours to represent all separating edge-pairs and all separating edge-triplets. Linear time algorithms for testing 3-edge-connectivity were given in Galil and Italiano [GI91] and Ramachandran [Ram90]. Using an approach that is different from ours, Watanabe, Yamakado and Onaga [WYO91] gave a linear time sequential algorithm for finding a smallest 3-edge-connectivity augmentation, but our algorithm appears to be simpler than theirs. For testing 4-edge-connectivity, Galil and Italiano [GI91] have a sequential algorithm runs in the same time complexity as ours. Gabow [Gab91] gave an $O(m + n \log n)$ time sequential algorithm for finding a smallest 4-edge-connectivity augmentation. Note that the augmented graph for reaching 3-edge-connectivity or 4-edge-connectivity produced by our algorithms (and all other algorithms mentioned here) is a multi-graph. We do not know how to find a smallest 3-edge-connectivity or 4-edge-connectivity augmentation if we want the resulting graph to be simple.

7.2 Definitions

Edge-Connectivity

An undirected graph G with more than one vertex is *k-edge-connected* if the graph obtained from G by removing any set of $k-1$ (or less) edges is still connected. The *edge-connectivity* of a graph G is k if G is k-edge-connected, but not $(k+1)$-edge-connected. A set of edges $\mathcal{A}$ with cardinality k is a *separating edge-set* if the removal of $\mathcal{A}$ disconnects G. A separating edge-set with cardinality 1 is a *cut edge*. A separating edge-set with cardinality 2 is a *separating edge-pair*. A separating edge-set with cardinality 3 is a *separating edge-triplet*. A *w-edge-connected component H* in a k-edge-connected graph G,

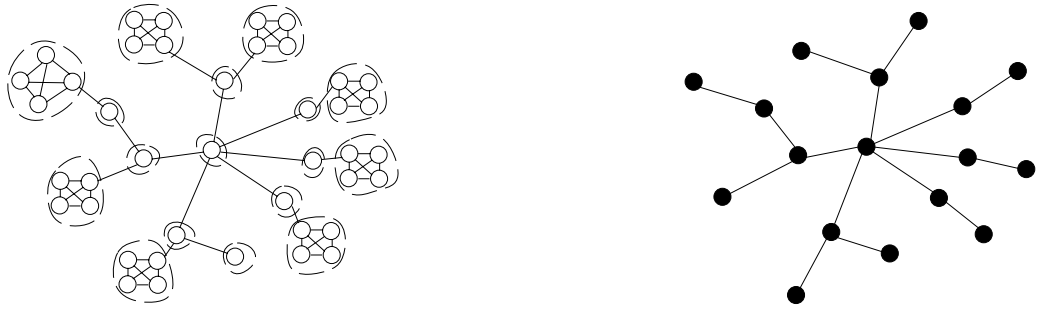


Figure 7.1: Illustrating a 2-edge-connected graph (on the left) and its 2-$eblk(G)$ (on the right). Note that the 2-$eblk(G)$ is a tree-like structure and vertex 10 is shared by three cycles.

where $w > k$, is a maximal subgraph of G such that H is w-edge-connected. Given a connected graph G, it is well-known [ET76] that we can build a 2-edge-block graph 2-$eblk(G)$ where each vertex represents either a vertex that is not in any 2-edge-connected component of G or the set of vertices in a 2-edge-connected component. There is an edge between two vertices v_1 and v_2 in 2-$eblk(G)$ if and only if there is a cut edge between the two components represented by v_1 and v_2. It is also well-known [ET76] that 2-$eblk(G)$ is a tree if G is connected.

The notation of a k-$eblk(G)$ for a $(k-1)$-edge-connected graph G is described in Dinits, Karzanov and Lomosonov [DKL76] (in Russian, a description is available in [NGM90]). They showed the following results. If k is even, k-$eblk(G)$ is a tree. If k is odd, k-$eblk(G)$ is a *tree-like structure* where two cycles in k-$eblk(G)$ share at most one vertex. An example of 3-$eblk(G)$ is shown in Figure 7.1. Edges not in a cycle are called *tree-edges* and edges in a cycle are called *cycle-edges*. Each vertex in G is mapped into exactly one vertex in k-$eblk(G)$. Each vertex in k-$eblk(G)$ represents an empty set, a vertex that is not in any k-edge-connected component of G, or the set of vertices in a

k-edge-connected component. Each tree-edge (a_1, a_2) in k-$eblk(G)$ represents a separating edge-set whose removal separates G into two components G_1 and G_2, where G_i contains the set of vertices represented by a_i, $i \in \{1,2\}$. Each cycle-edge (b_1, b_2) represents the set of edges of cardinality $\frac{k-1}{2}$ between the component represented by b_1 and the component represented by b_2. The union of two sets of edges represented by two cycle-edges in the same cycle is a separating edge-set. Each of the separating edge-sets in G is either represented by a tree-edge in k-$eblk(G)$ or two cycle-edges in the same cycle. Let G_1 and G_2 be the two components obtained from G by the removal of a separating edge-pair and let k-$eblk(G_1)$ and k-$eblk(G_2)$ be the two components obtained from e-$blk(G)$ by the removal of its corresponding tree-edge or two cycle-edges. Vertices in G_i, $i = 1$ and 2, map into vertices in k-$eblk(G_i)$, respectively. The structure k-$eblk(G)$ can be computed in $O(nm)$ time for arbitrary k [KT86].

Singular-Set

A subset of l vertices V' in a k-edge-connected graph $G = (V, E)$ is an *l-singular set* if there exists a set of k edges in G whose removal separates G into two components, one of which consists of exactly the vertices in V'.

7.3 Finding All Separating Edge-k-Sets, $k \in \{2,3\}$

In this section, we develop algorithms to form 3-$eblk(G)$ for an undirected graph G that is 2-edge-connected and to form 4-$eblk(G)$ when G is 3-edge-connected.

7.3.1 Finding All Separating Edge-Pairs

Let g be a 2-edge-connected graph. We first construct its 2-block graph 2-$blk(G)$ by using the algorithm in Chapter 3 and [ET76]. Note that

each 2-block in 2-$blk(G)$ represents either a vertex whose adjacent edges are all cut edges or a biconnected component. For each 2-block H in G that is not a single vertex, we construct its 3-block graph 3-$blk(H)$ by using the algorithm in Chapter 4 and [HT73, FRT93]. Note that in 3-$blk(H)$, β-vertices correspond to triconnected components or a vertex of degree 2 in H, σ-vertices correspond to separating vertex-pairs, and π-vertices correspond to polygons.

The following lemma gives a method to find all separating edge-pairs in a 2-edge-connected graph G by examining cutpoints, 1-singular sets, Tutte pairs (defined in Chapter 4), 2-singular sets, and polygons (defined in Chapter 4) in the 3-block trees for all 2-blocks (defined in Chapter 3) in G.

Lemma 7.3.1 *Let G be a 2-edge-connected graph and let $\{(a_1, a_2), (b_1, b_2)\}$ be a separating edge-pair in G. Then at least one of the following is true.*
(i) $\{a_1, a_2, b_1, b_2\} = \{c_1, c_2\}$, where c_i is either a cutpoint or a 1-singular set, $i \in \{1, 2\}$ (e.g., $a_1 = b_1 = c_1$ and $a_2 = b_2 = c_2$).
(ii) $\{a_1, a_2, b_1, b_2\} = \{c, s_1, s_2\}$, where c is either a cutpoint or a 1-singular set in G and $\{s_1, s_2\}$ is either a Tutte pair or a 2-singular set in G (e.g., $a_1 = a_2 = c$, $b_1 = s_1$ and $b_2 = s_2$).
(iii) (a_1, a_2) and (b_1, b_2) are two edges in a polygon represented by a π-vertex in 3-$blk(G)$, and $\{a_1, a_2\}$ and $\{b_1, b_2\}$ are not separating vertex-pairs.

Proof. Note that there are only connected components in the resulting graph obtained from G by removing any one separating edge-pair. Let G_1 and G_2 be the two connected components in $G - \{(a_1, a_2), (b_1, b_2)\}$. Assume that vertices a_i and b_i are in G_i, $i \in \{1, 2\}$. If there exists an i^*, $i^* \in \{1, 2\}$, such that $a_{i^*} = b_{i^*} = c$ and G_{i^*} contains more than one vertex, then either c is a cutpoint or a 1-singular set. If G_{i^*} contains a single vertex c, then c is degree-2 (i.e., a

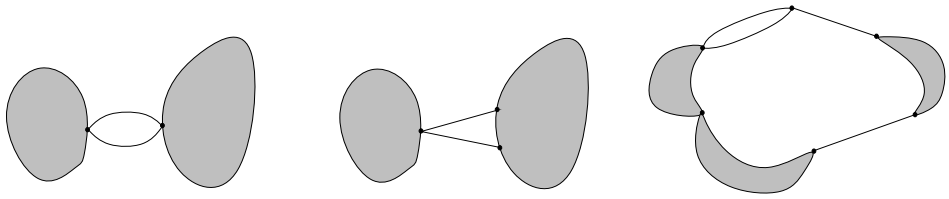


Figure 7.2: Illustrating the three types of separating edge-pairs that we have in a 2-edge-connected graph. Note that in Figure 7.2.(c), $\{1,2,3,4,5,6\}$ is a polygon in 3-$blk(G)$. In this polygon, edge (5, 6) is not solid; edges (2, 3) and (4, 5) are 1-solid; edge (1, 2) is 2-solid.

1-singular set). If $\{(a_1,a_2),(b_1,b_2)\} = \{c,s_1,s_2\}$ and $s_1 \neq s_2$, then (s_1, s_2) is either a 2-singular set or a separating pair. From the definition of a polygon it is easy to see that if (s_1, s_2) is a separating pair in a polygon $\mathcal{C}$, then s_1 and s_2 are not adjacent in $\mathcal{C}$. Thus (s_1, s_2) is a Tutte pair. Otherwise (a_1, a_2) and (b_1, b_2) must be two edges in a polygon obtained by decomposing G into triconnected components. (For details, see [HT73, FRT93, Ram93].) □

An example is shown in Figure 7.2.

Let G be a biconnected graph and let 3-$blk(G)$ be its 3-block graph. We define an edge (a_1, a_2) in a polygon represented by a π-vertex in 3-$blk(G)$ to be *i-solid* if there are exactly i multiple edges between vertices a_1 and a_2 in G and $\{a_1, a_2\}$ is not a separating vertex-pair in G. Examples are shown in Figure 7.2.(c). The following corollary of Lemma 7.3.1 shows the relation between a polygon in a 3-$blk(G)$ and a cycle in 3-$eblk(G)$.

Corollary 7.3.2 *Let G be a biconnected graph and let C be a polygon represented by a π-vertex in 3-$blk(G)$. If C contains at least two 1-solid edges, then any two 1-solid edges in C is a separating edge-pair for G.* □

Note that if a polygon C contains exactly two 1-solid edges e_1 and e_2, then $\{e_1$, $e_2\}$ is a separating edge-pair corresponding to a tree-edge in 3-$eblk(G)$. Note

also that a polygon C with more than two 1-solid edges corresponds to a cycle in 3-$eblk(G)$ that consists of all of the 1-solid edges in C.

From Lemma 7.3.1 and Corollary 7.3.2, we have the following theorem.

Theorem 7.3.3 *There is a linear time sequential algorithm for building k-eblk(G) for an undirected graph G with edge-connectivity $k-1$, $\forall k$, $1 \leq k \leq 3$. There is also a parallel algorithm for this problem that runs in $O(\log n)$ time using $O(\frac{(n+m)\log\log n}{\log n})$ processors on a CRCW PRAM, given where n and m are the number of vertices and edges in G, respectively.*

Proof. Note that there are $O(n)$ cutpoints in G. The number of Tutte pairs is $O(n)$. The number of 1-singular sets is $O(n)$ and these sets are represented as trivial 2-blocks. The number of 2-singular sets in a 2-edge-connected graph is $O(n)$. They can can found by checking polygons with 3 vertices in 3-$blk(G)$. We can find all separating edge-pairs according to Lemma 7.3.1 by building a table for all 1-singular sets, cutpoints, Tutte pairs, 2-singular sets, and using a bucket sort routine. □

7.3.2 Finding All Separating Edge-Triplets

Let G be a 3-edge-connected graph. We first construct its 2-block graph 2-$blk(G)$ by using the algorithm in Chapter 3. For each 2-block H in G that is not a single vertex, we construct its 3-block graph 3-$blk(H)$ by using the algorithm in [HT73, FRT93, Ram93]. For each 3-block I in a 2-block H that is triconnected, we construct its 4-block graph 4-$blk(I)$ by using the algorithm in Chapter 6 and [KTDBC91].

The following lemma gives a method to find all separating edge-triplets in a 3-edge-connected graph G by examining cutpoints, 1-singular sets,

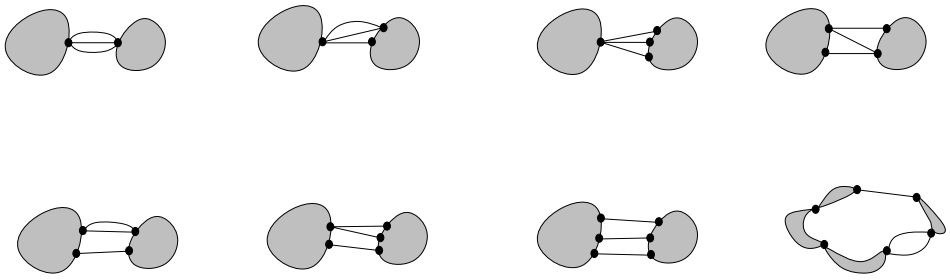


Figure 7.3: Illustrating the eight types of separating edge-pairs that we have in a 3-edge-connected graph.

Tutte pairs, 2-singular sets, flowers, 3-singular sets and polygons in the 3-block trees for all 2-blocks in G. The proof of this lemma is similar to the proof for Lemma 7.3.1.

Lemma 7.3.4 *Let G be a 3-edge-connected graph and let $S = \{(a_1, a_2), (b_1, b_2), (c_1, c_2)\}$ be a separating edge-triplet in G. Let G_1 and G_2 be the two connected components in $G - \{(a_1, a_2), (b_1, b_2), (c_1, c_2)\}$. Assume that vertices a_i, b_i, and c_i are in G_i, $i \in \{1, 2\}$. Then at least one of the following is true.*
(i) $\{a_i, b_i, c_i\}$ is a cutpoint, 1-singular set, Tutte pair, 2-singular set, flower, or 3-singular set, $\forall i$, $i \in \{1, 2\}$.
(ii) (a_1, a_2) is a 1-solid edge in a polygon C, $(b_1, b_2) = (c_1, c_2)$ (i.e., there are two multiple edges between vertices b_1 and b_2), and (b_1, b_2) is a 2-solid edge in the same polygon C. □

An example is shown in Figure 7.3.

Corollary 7.3.5 *Let G be a biconnected graph and let C be a polygon represented by a π-vertex in 3-blk(G). If C contains exactly one 1-solid edge and at least one 2-solid edge, then the combination of the 1-solid edge and any one 2-solid edge in C is a separating edge-triplet for G.* □

From Lemma 7.3.4 and Corollary 7.3.5, we have the following theorem.

Theorem 7.3.6 *There is an $O(n\alpha(m,n)+m)$ time sequential algorithm for building 4-eblk(G) for an undirected graph G that is 3-edge-connected.*

Proof. The proof of this theorem is similar to the proof for Theorem 7.3.3 with the only exception that we need to find all 3-singular sets. These 3-singular sets can be found by checking all 3-blocks in G that are complete graphs of 4 vertices if G contains more than 6 vertices. It is easy to check 3-singular sets in a graph with no more than 6 vertices. □

7.4 Edge-Connectivity Augmentation

The approach used in [NGM90] to increase the edge-connectivity of any graph G by δ is to first develop an algorithm for increasing the edge-connectivity of G by 1. By choosing appropriate edges, [NGM90] further showed that it is possible to optimally increase the edge-connectivity of any graph by δ by applying the basic algorithm δ times. Their algorithm for increasing the edge-connectivity of any graph by 1 is as follows. (For $k = 2$, this specializes to the linear time algorithm of [ET76].) Let G be the input graph with edge-connectivity $k-1$. We first compute k-*eblk*(G). A *leaf* in k-*eblk*(G) is defined as either a degree-1 vertex or a degree-2 vertex that is in a cycle. (Note that the set of vertices in G which corresponds to a leaf is called an *extreme set* in [NGM90].) We perform the following modified depth-first search (DFS) starting from an arbitrary vertex in k-*eblk*(G). If the algorithm is visiting a vertex that is not in a cycle, then we perform the normal DFS. If the algorithm visits a vertex v in a cycle C from another vertex in C, then the

algorithm must visit all adjacent unvisited vertices of v that are not in C before visiting its unvisited adjacent vertex in C. Leaves in k-$eblk(G)$ are numbered according to the orders they are encountered during the modified DFS. Let l be the number of leaves and let L_i be the ith leaf. It is shown in [NGM90] that we can increase the edge-connectivity of G by 1 by adding an edge between L_i and $L_{i+\lfloor \frac{l}{2} \rfloor}$, $\forall i$, $1 \leq i \leq \lceil \frac{l}{2} \rceil$. The above algorithm can be implemented in linear time given k-$eblk(G)$. For details, see [NGM90].

To guarantee that the edge-connectivity of a graph can be optimally increased by δ by applying the above basic algorithm δ times, we must pick an appropriate vertex in each leaf on which a new incoming edge can be incident. We will call such a vertex *a demanding vertex*. Let λ be the current edge-connectivity and let $\lambda+\delta$ be the desired edge-connectivity. Let H_i, $1 \leq i \leq \delta$, be a subgraph of H_{i-1} (where $H_0 = G$) which corresponds to a leaf (or an isolated vertex) in $(\lambda+i)$-$eblk(H_{i-1})$ and the out-degree of vertices in H_i is at most $\lambda+i$. (Note that we extend the definition of an edge-block graph for an input graph G in a way that if G is a single vertex than its edge-block graph is a single vertex.) It is shown in [NGM90] that if we pick a vertex in H_δ, then we can maintain the overall optimality by applying the basic algorithm δ time. (Note that this type of strategy will not work for increasing the vertex-connectivity of an undirected graph. For details, see Chapter 5.) For $\lambda \in \{0,1,2,3\}$, we can pick such a demanding vertex for each leaf in λ-$eblk(G)$ in linear time. Thus we can find a smallest $(\lambda-1)$-edge-connectivity augmentation in linear time given λ-$eblk(G)$.

We now show how to implement the above algorithm efficiently in parallel given the edge block tree. We first locate the set of vertices $\mathcal{S}$ that are shared by at least two cycles in k-$eblk(G)$. For each vertex v in $\mathcal{S}$, we create a

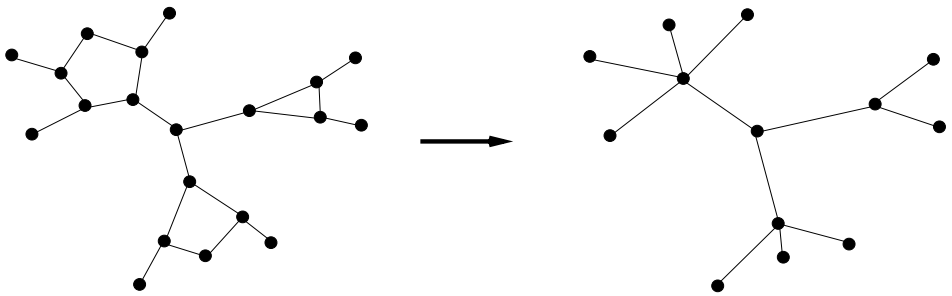


Figure 7.4: Illustrating the transformation of a tree-like structure shown in Figure 7.1.(b) into a tree. The original 3-$eblk(G)$ (Figure 7.1.(b)) is first transformed into a graph (Figure 7.4.(a)) such that two cycles do not share any vertex by duplicating vertices that are in more than one cycle. We then contract vertices of degree at least 3 in a cycle into a single vertex. The resulting graph (Figure 7.4.(b)) is a tree.

new vertex v_i for the ith cycle $C_{v,i}$ that contains v. Edges adjacent to v in $C_{v,i}$ are reconnected to v_i. We also connect v to each v_i. Thus no cycle shares a vertex with another cycle after the above step. For each cycle C, we contract vertices of degree at least 3 in C into a single vertex u_C with the requirement that the order of the edges adjacent to u_C is the same as the order of the edges we encountered when we traverse C clockwise starting from an arbitrary vertex. After removing multiple edges, the resulting graph is a tree. Note that leaves in the original tree-like structure become leaves in the resulting graph after the transformation. An example is shown in Figure 7.4.

We can perform a preorder numbering on leaves in the resulting tree. All of the above steps (including the one for finding demanding vertices) can be implemented in $O(\log n)$ time on a CRCW PRAM using $O(\frac{(n+m)\log\log n}{\log n})$ processors. The additional steps described in [NGM90] in order to guarantee the overall optimality if we want to apply the basic algorithm more than once are easily parallelized. Thus we have the following theorems.

Theorem 7.4.1 *There is a linear time sequential algorithm for finding a smallest set of edges whose addition 3-edge-connects an undirected graph. There is also a parallel algorithm for solving this problem in $O(\log n)$ time using $O(\frac{(n+m)\log\log n}{\log n})$ processors on a CRCW PRAM, where n and m are the number of vertices and edges in G, respectively.* □

Theorem 7.4.2 *There is an $O(n\alpha(m,n)+m)$ time sequential algorithm for finding a smallest set of edges whose addition 4-edge-connects an undirected graph.* □

7.5 Concluding Remarks

We have presented a simple algorithm to construct a compact representation for all separating edge-pairs in a graph G that is not 3-edge-connected by using compact representations for all cutpoints and separating vertex-pairs in G. We can also use this algorithm to find a smallest 3-edge-connectivity augmentation. Our algorithm runs in linear time sequentially. Its parallel version runs in $O(\log n)$ time on a CRCW PRAM using $O(\frac{(n+m)\log\log n}{\log n})$ processors given any reasonable sparse representation of the input graph. We also have shown a simple linear time algorithm to construct a compact representation for all separating edge-triplets in a graph G that is not 4-edge-connected given compact representations for all cutpoints, separating vertex-pairs, and separating vertex-triplets in G. We can also use this algorithm to find a smallest 4-edge-connectivity augmentation in the same complexity as the algorithm for building a compact representation for all separating vertex-triplets [KTDBC91] (which runs in in $O(n\alpha(m,n)+m)$ time sequentially). We do not have an efficient parallel algorithm for this problem since there is no efficient parallel algorithm

known for finding a compact representation for all separating vertex-triplets in a graph.

Note that all but two of the parallel routines used in this chapter run in $O(\log n)$ time on an EREW PRAM using $O(\frac{n+m}{\log n})$ processors. One exception is the parallel routine for computing connected components, which runs in $O(\log n)$ time on a CRCW PRAM using $O(\frac{(n+m)\alpha(m,n)}{\log n})$ processors [CV86] given the adjacency list of the input graph, where α is the inverse Ackermann function. This routine can also be implemented on an EREW PRAM in $O(\log n \log\log n)$ time using $O(n+m)$ processors [CL93]. The other exception is the parallel routine for performing bucket sorting, which runs in $O(\log n)$ time on a CRCW PRAM using $O(\frac{(n+m)\log\log n}{\log n})$ processors [CV86]. This routine can also be implemented on an EREW PRAM in $O(\log n)$ time using a linear number of processors. Thus algorithms reported in this chapter can be implemented on an EREW PRAM that run in $O(\log n \log\log n)$ time using a linear number of processors giving any reasonable sparse representation of the input graph.

Chapter 8

Implementation of Augmentation Algorithms

In this chapter, we describe our implementation of the linear time sequential algorithms for finding a smallest 2-edge-connectivity augmentation [ET76] and for finding a smallest biconnectivity augmentation which is given in Chapter 3. We also describe our implementation of an efficient parallel algorithm [ET76, Sor88] for finding a smallest 2-edge-connectivity augmentation. In addition to describing our implementation, we also provide performance data for our code.

8.1 Sequential Implementation

To speed up our sequential implementations, we used the graph manipulation package NETPAD [DMM92]. (An introduction of NETPAD is given in Part II of this thesis.) The NETPAD software contains a rich set of fundamental graph algorithms that can be used to implement complex graph algorithms. We tested our implemented algorithms on a SUN SPARC 10/41.

The organization of this section is as follows. In Section 8.1.1, we describe the implementation and the performance data of the smallest 2-edge-connectivity augmentation algorithm given in [ET76]. In Section 8.1.2, we describe the implementation and the performance data of the smallest biconnectivity augmentation algorithm given in Chapter 3.

8.1.1 Smallest Two-Edge-Connectivity Augmentation

We implemented the linear time sequential algorithm for finding a smallest 2-edge-connectivity augmentation given in [ET76] (which is briefly described in Section 7.4). After coding, we tested the performance of our implementation. Since the performance depends on the number of edges added, we needed to generate test graphs whose augmentation numbers are approximately known.

Note that the 2-edge-connectivity augmentation number of a tree is $\left\lceil \frac{\ell}{2} \right\rceil$, where ℓ is the number of leaves in the tree. Since the input graph is converted into its 2-edge-block tree data structure before applying the augmentation algorithm, we tested our program on trees. We generated our test graph (tree) with about ℓ leaves and about w internal nodes using the following method. We first generated an empty graph with $\ell+w$ isolated vertices. Let the first ℓ vertices be leaf-nodes and let the rest of the vertices be internal-nodes. We randomly connected a degree-0 leaf-node with an arbitrary internal-node until all leaf-nodes were connected. We then randomly connected a degree-0 or degree-1 internal-node with an arbitrary internal-node such that no cycle was created. Note that the degree of some internal-nodes remained to be one after we made all feasible connections. By doing this, we created a tree with a total of $\ell + w$ nodes and at least ℓ leaves. Our experiments showed that for $w = 2\ell$, the graphs that we constructed contained about 1.1ℓ leaves. The number of internal nodes is reduced since some of them might be degree 1 after our construction.

We tested our program on trees generated with $w = 2\ell$. For each value of w, we tested four different graphs and recorded the average running time. The performance data for running our program on a SPARC 10/41 workstation is shown in Table 8.1.

w	ℓ	test 1 (secs)	test 2 (secs)	test 3 (secs)	test 4 (secs)	average (secs)
500	250	0.07	0.07	0.07	0.05	0.06
1,000	500	0.13	0.10	0.13	0.12	0.12
1,500	750	0.20	0.18	0.18	0.22	0.20
2,000	1,000	0.25	0.25	0.27	0.25	0.25
2,500	1,250	0.33	0.32	0.33	0.35	0.33
3,000	1,500	0.40	0.40	0.42	0.40	0.40
3,500	1,750	0.48	0.47	0.53	0.48	0.49
4,000	2,000	0.57	0.57	0.57	0.57	0.57
4,500	2,250	0.62	0.63	0.65	0.62	0.63
5,000	2,500	0.70	0.72	0.72	0.73	0.72
5,460	2,730	0.93	0.80	0.80	0.82	0.84

Table 8.1: Performance data for finding a smallest 2-edge-connectivity augmentation on a SPARC 10/41.

8.1.2 Smallest Biconnectivity Augmentation

The coding of our linear time sequential algorithm for finding a smallest biconnectivity augmentation given in Chapter 3 required a little more work as we had to implement several non-trivial data structures. After coding, we tested the performance of our implemented algorithm. Since the performance of our program depends on the number of edges added, we needed to test our program on input graphs whose biconnectivity augmentation numbers are known approximately.

Note that the biconnectivity augmentation number depends on the largest degree among all cutpoints and the number of leaves in the 2-block tree. Note also that in a 2-block tree, internal nodes consist of cutpoints and 2-blocks. We generated a tree-like structure whose 2-block tree consisted of about ℓ leaves, c cutpoints and w internal 2-blocks using the following method. We begin by first generating a tree T with about ℓ leaves and $c + w$ internal nodes by the following steps. Let L be a set of ℓ isolated vertices, C be a set of c isolated vertices, and W be a set of w isolated vertices. We first generated

an empty graph with the set of vertices in $L \cup C \cup W$. We randomly connected a degree-0 vertex in L with a vertex in C until all vertices in L were degree 1. We then randomly connected a vertex in C with a vertex in W as long as no cycle was created. The resulting graph was the tree T that we needed with about ℓ leaves and about $c + w$ internal nodes. Our experiments with $\ell = w = c$ showed that the average number of leaves in T was about 1.2ℓ. The number of cutpoints is reduced since some of them might be degree 1 after our construction.

Using the following method we then converted T into a tree-like structure G such that the 2-block graph of G is isomorphic to T. For a vertex v in W with degree > 1, let $N(v)$ be the set of its neighbors. We created a simple cycle by linking vertices in $N(v)$ one after the other. After adding such a cycle to T for each vertex with degree > 1 in W, we created a tree-like structure whose 2-block tree is isomorphic to T.

We tested our programs on tree-like structures generated with $w = c = \ell$. For each value of w, we tested four different graphs and recorded the average running time. The performance data for running our program on a SPARC 10/41 workstation is shown in Table 8.2.

8.2 Parallel Implementation

We implemented the simple PRAM algorithm for finding a smallest 2-edge-connectivity augmentation given in [ET76, Sor88] on a massively parallel SIMD computer MasPar MP-1 [Mas92d]. (An introduction of the MasPar is given in Part II of this thesis.) This algorithm runs in $O(\log n)$ time on a CRCW PRAM using a linear number of processors given the adjacency list of the input graph, where n is the number of vertices in the input graph. The

c	w	ℓ	test 1 (secs)	test 2 (secs)	test 3 (secs)	test 4 (secs)	average (secs)
200	200	200	0.08	0.08	0.07	0.08	0.08
400	400	400	0.20	0.20	0.17	0.20	0.19
600	600	600	0.27	0.25	0.28	0.28	0.27
800	800	800	0.38	0.38	0.40	0.40	0.39
1,000	1,000	1,000	0.52	0.50	0.48	0.48	0.50
1,200	1,200	1,200	0.65	0.60	0.57	0.58	0.60
1,400	1,400	1,400	0.80	0.77	0.70	0.73	0.75
1,600	1,600	1,600	0.85	0.82	0.90	0.83	0.85
1,800	1,800	1,800	0.98	1.00	1.00	1.02	1.00
1,975	1,975	1,975	1.08	1.17	1.10	1.13	1.12

Table 8.2: Performance data for finding a smallest biconnectivity augmentation on a SPARC 10/41.

MasPar computer that we used had 16,384 processors where each processor is about 230 times slower than a SUN SPARC 10/41. We used the parallel language MPL [Mas92b, Mas92c] that is an extension of the C programming language [KR88]. (An introduction of the MPL is also given in Part II of this thesis.) Since the MPL does not support the use of virtual processors, we implemented the algorithm only to handle the case when the input size is no more than the number of physical processors in the MasPar (i.e., 16,384).

After coding, we tested our programs using the test graphs generated by the method described in Section 8.1.1. For each value of w, we tested four different graphs and recorded the average running time. The performance data is shown in Table 8.3.

From Tables 8.1 and 8.3, we notice that although our parallel program had a slower rate of increase in running time than our sequential algorithm, our parallel algorithm actually ran slower (in real time) on the largest inputs. It may be the case that by allocating more processors, our parallel program could run faster. In Part II, we address the issue of using virtual processors in parallel programs.

w	ℓ	test 1 (secs)	test 2 (secs)	test 3 (secs)	test 4 (secs)	average (secs)
500	250	0.62	0.61	0.61	0.61	0.61
1,000	500	0.78	0.84	0.82	0.76	0.80
1,500	750	0.85	0.94	0.90	0.85	0.88
2,000	1,000	1.02	1.03	0.91	1.03	1.00
2,500	1,250	1.10	1.18	1.27	1.09	1.16
3,000	1,500	1.14	1.14	1.22	1.08	1.15
3,500	1,750	1.40	1.41	1.30	1.36	1.37
4,000	2,000	1.65	1.38	1.52	1.47	1.50
4,500	2,250	1.57	1.37	1.49	1.50	1.48
5,000	2,500	1.59	1.76	1.57	1.63	1.64
5,460	2,730	1.66	1.60	1.64	1.65	1.64

Table 8.3: Performance data for finding a smallest 2-edge-connectivity augmentation in parallel on the MasPar MP-1 with 16,384 processors.

Chapter 9

Conclusion and Open Problems

9.1 Summary

In Part I, we have presented several algorithms for finding a smallest k-vertex-connectivity augmentation, where $k \leq 4$. We also have given an efficient method to transform the problem of finding a smallest edge-connectivity augmentation into the problem of finding a smallest vertex-connectivity augmentation. The approach used by our algorithms for finding a smallest vertex-connectivity augmentation is to first consider the case when the input graph G is $(k-1)$-vertex-connected. We derived a data structure based on the structure of G to describe all necessary information needed for our augmentation algorithm. This data structure represents all separating k-sets and all maximal subsets of vertices that are k-vertex-connected. For example, for $k = 2$, we used the well-known block tree structure [Har69, Tut66]. For $k = 3$, we modified the well-known tree of triconnected components [Tut66, HT73] to obtain the 3-block tree. For $k = 4$, the data structure was the 4-block tree [KTDBC91].

We observed that there is a natural correspondence between the augmentation of the input graph and the augmentation of its k-block tree. Using this property, our augmentation algorithm worked on the derived data structure instead of working on the original graph. We first defined a lower bound for the augmentation number based on the structure of the k-block tree. Then we proved that this lower bound could be reduced by one by properly adding an edge. We then gave an efficient implementation of this addition of edges.

By doing this, we have given an efficient algorithm for finding a smallest augmentation.

The case when the input graph is not $(k-1)$-connected required more work. Basically, we first extended the k-block graph data structure for an input graph that is not $(k-1)$-vertex-connected by a recursive decomposition to obtain an i-block graph from each $(i-1)$-block in an $(i-1)$-block graph, $1 < i \leq k$. Then we derived a simple lower bound for the augmentation number and gave an inductive proof to show that this given lower bound is always achievable. We were able to obtain efficient sequential and parallel algorithms to find such an augmentation for the case when $k = 3$. We feel that a similar approach can be used to extend our result for $k = 4$. Although the algorithm for the case $k = 3$ was simple, the proof of correctness was rather involved.

9.2 Open Problems

Although a polynomial time approximation algorithm is known for finding a smallest k-vertex-connectivity augmentation on a $(k-1)$-vertex-connected graph, for an arbitrary k [Jor93b], the problem remains open for $k \geq 5$ if one needs to solve the problem exactly in polynomial time. We feel that the strategy derived in this thesis might be useful in solving this problem. For this, further insights into the structure of a k-vertex-connected graph, $k \geq 4$, (e.g., [CBKT93, Kan88, Mat72, Mat76, Mat78]) will be needed.

Part II

Implementation of Efficient Parallel Graph Algorithms

Chapter 10

Preliminaries

Graphs play an important role in modeling the underlying structure of many real-world problems. Over the past couple of decades, efficient sequential algorithms have been developed for several graph problems and have been implemented on sequential machines (e.g., [DMM92, MN89]). The NETPAD system [DMM92] at Bellcore is a general tool for graph manipulations and algorithm design that facilitates such implementations. More recently, several research results on efficient parallel algorithms have been developed [JáJ92, KR90, Lei92, Qui87], not much implementation has been done. In Part II, we describe our work on implementing efficient PRAM graph algorithms on a massively parallel SIMD computer, the MasPar MP-1.

The organization of this chapter is as follows. Section 10.1 gives a description of our implementation strategy. Section 10.2 describes the MasPar hardware and software. Section 10.3 describes the set of graph algorithms that we have implemented and the strategy we used in implementing them. Section 10.4 describes the mapping between the PRAM model and the MasPar. Section 10.5 gives a brief overview of the rest of the chapters in Part II.

10.1 Implementation Strategy

Several strategies can be used to implement parallel algorithms on a parallel computer. One possible strategy is to implement different algorithms for different architectures. Since parallel machines are widely diverse in their

architectures, one can take advantage of the special properties offered by an architecture and fine-tune the algorithms to run well on a particular machine. For parallel algorithms using this approach, see [Lei92, Qui87]. However, this time-consuming process must be carried out each time a new architecture arrives. This approach may be useful for some of the very important subroutines used in the machine (e.g., sorting [BLM+91, PS90]). However, for complicated combinatorial problems, reinventing different algorithms for different architectures tends not to be a feasible solution. As the problems get more complicated, it takes longer time to derive efficient algorithms. Further, we feel that this is not a very good strategy as one often discovers that the fundamental algorithmic techniques underlying the parallel algorithms for most problems are independent of the particular parallel machine being used. Thus one should utilize these basic techniques to assist the implementation of parallel algorithms.

In view of the above, a natural strategy is to use parallel algorithms developed on an abstract parallel machine model. Several abstract models that are closely related to real parallel machine architectures have been proposed [Ble89, DNS81, GMR93, HS86, Sch80, CKP+93]. Instead of using a new model, we have performed a direct implementation of parallel algorithms based on the popular PRAM model [JáJ92, KR90, Rei93]. Although the PRAM is an idealized theoretical model that does not capture the real cost of performing inter-processor communications on the MasPar, we believe that it provides a good abstract model for developing parallel algorithms. Parallel algorithms developed on the PRAM model are often very modular in structure (or have parallel primitives). Problems are solved by calling these parallel primitives. For solving undirected graph problems, a set of parallel primitives required for constructing an ear decomposition has proved to be very useful [Ram93, Vis91].

Our parallel implementation follows this approach. We first built a kernel which consists of commonly used routines in parallel graph algorithms. Then we implemented efficient parallel graph algorithms developed on the PRAM model by calling routines in the kernel.

Our experience with implementing PRAM graph algorithms on the MasPar MP-1 as will be reported in Chapters 11, 12, and 13 supports our viewpoint that efficient PRAM algorithms are adaptable to run on real machines. The basic primitives should be fine-tuned for the real machine (or use algorithms whose time and processor complexities have been throughout analyzed on a more accurate model, e.g., [CKP+93]), but the overall structure of a complex PRAM algorithm can be mapped directly on to the real machine.

10.2 Programming Environment

The MasPar computer [Mas91c] is a fine-grained massively parallel single-instruction-multiple-data (SIMD) computer. All of its parallel processors synchronously execute the same instruction at the same time. A simplified version of its architecture is shown in Figure 10.1.

The MasPar has a front end processor running the Unix operating system [RT74] and a Data Parallel Unit (DPU) for execution of parallel programs. The front end machine is a micro-VAX workstation. The DPU consists of an Array Control Unit (ACU) and 16,384 Processor Elements (PE's). The ACU is a special purpose processor for controlling the execution of all of the PE's. Programs are stored in one special local memory bank of ACU and broadcast to each PE simultaneously. The architecture of the MasPar allows very efficient broadcasting from the ACU to all PE's. Since the ACU is about 10 times faster than each individual PE [Mas91c], the other purpose of the ACU is to

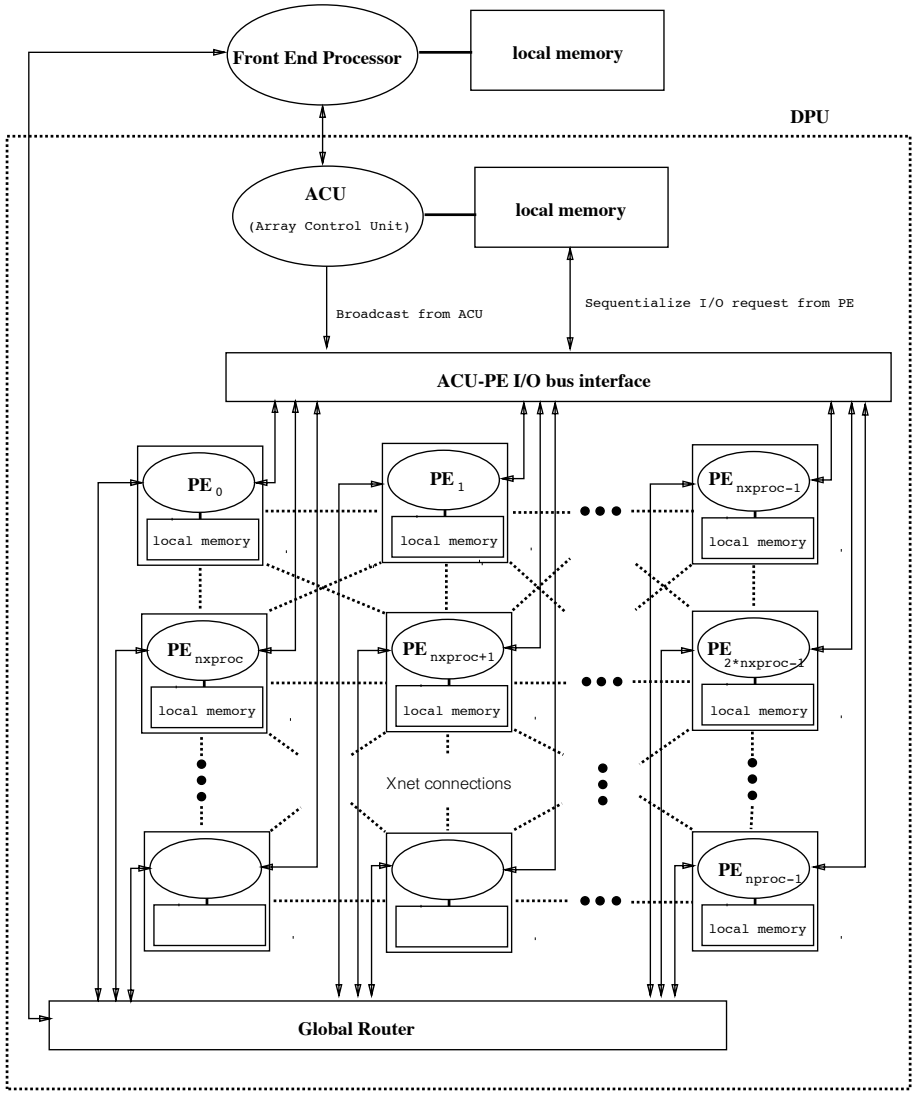


Figure 10.1: System architecture for the MasPar computer.

perform simple local computations and broadcast the results to all PE's. The ACU is a special processor that is designed for load and save operations, and hence the ACU might not be very efficient in computing complex arithmetic operations. For performing global arithmetic operations, it appears that the preferred method is to perform the computations on the front end processor and have the front end transfer the results back to the ACU.

All of the PE's are organized as a two-dimensional matrix. In Figure 10.1, $nxproc = 128$ and $nproc = 16{,}384$ for our machine, but these numbers may be different at other installations. Each PE consists of a special processor and a bank of local memory (about 64 kilobytes in our system). Upon receiving an instruction from the ACU, the processor will execute the instruction on its own local data. Each PE is connected to its 8 neighbors (toroidal wrapped) in a connection scheme called XNET. All PE's are also connected via a global router. Local data inside each PE can be exchanged through the global router as well as the local XNET. The XNET configuration is faster, but requires that all PE's receving data from the same direction. For transmitting a 32-bit data through distance 1, the XNET communication is slightly less than 100 times faster than the time needed to go through the global router [Mas91b]. During the computation, it might be necessary for the PE's to access the local memory of the ACU (about 1 megabytes). Such I/O requests are sequentialized and carried out one at a time. This process is very time-consuming.

The MasPar system provides a fast way of transferring data between the local memory of the front end and the local memory of the ACU. It also provides a fast way of transferring data between the local memory of the front end and the local memory banks of the PE's. In order to perform the latter transfer, the set of PE's to receive data from the front end must form a rectangular block. More details are described in Chapter 11.

We used the MasPar Parallel Language (MPL) [Mas91a, Mas91b] to implement our algorithms. MPL is an extension of the C language described in [KR78]. In addition to all of the standard C language features, it allows the user program a set of PE's to execute the same instruction on their own local data. MPL leaves the responsibility of processor allocation to the programmer. During the execution of the MPL program, the programmer must specify the set of PE's that are to execute the current instruction. MPL also allows the user to instruct the ACU to do local computations. MPL takes two types of input files. The first is a program file with a suffix ".c" which indicates that it is a pure C language program. MPL compiles this program and generates executable code that can be run on the front end. The second file is a program file with a suffix ".m" which indicates that it is a C language program that includes extended features for doing operations on the PE's. MPL compiles the program and generates executable code that can be run on the DPU. In the ".m" program, one can allocate local data on each PE by adding the keyword *plural* to a C data declaration statement. Variables declared without the *plural* keyword are allocated on the ACU. Any expression that involves a plural variable is computed on each active PE. An expression that contains no plural variables is computed on the ACU.

Using the MPL programming language, each PE can perform operations on 32-bit words and also on 64-bit words. (A 64-bit word can be declared by using the MPL data type *long long*.)

10.3 Parallel Graph Algorithms

In designing sequential graph algorithms, depth-first search and breadth-first search have been used as basic search strategies for solving various graph problems [Tar72]. Unfortunately, no efficient parallel algorithms are

known presently for these two search techniques [KR90]. Hence we are unable to obtain efficient parallel algorithms by parallelizing sequential algorithms based on depth-first search or breadth-first search. Instead, an alternative search technique called ear decomposition has proved to be a very useful tool for designing parallel graph algorithms [KR91a, KR90, MSV86, MR92, Ram93, RR89]. Combined with an efficient parallel routine for finding connected components [AS87] and the Euler tour technique [TV85], this gives efficient parallel algorithms for several important problems on undirected graphs which include various connectivity problems [KR91a, FRT93, MR92, Ram93], st-numbering [MSV86], planarity testing and embedding [RR89], finding a strong orientation and finding a minimum cost spanning forest. Figure 10.2 illustrates the building blocks for designing parallel graph algorithms using ear decomposition, the Euler tour technique and the routine for finding connected components.

Our parallel implementations followed this approach. We first built a kernel which consists of commonly used routines in parallel graph algorithms. Then we implemented efficient parallel graph algorithms developed on the PRAM model by calling routines in the kernel.

10.4 Mapping of the PRAM Model onto the MasPar Architecture

We map part of the local memory in each PE and the local memory of the ACU onto the PRAM global memory. The major difference between the PRAM model and the MasPar architecture is the time spent on global memory accessing. The MasPar allows constant time broadcasting of the contents of any single local memory location from the ACU to all PE's. However, it takes $O(P)$ time for P PE's to access the local memory of the ACU. Hence it is not efficient to map the global memory in the PRAM model to the local memory of

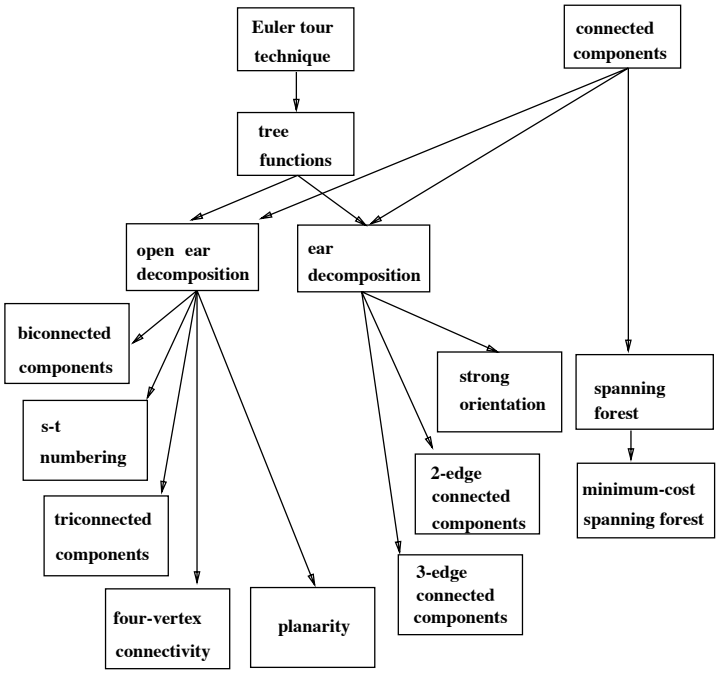


Figure 10.2: Parallel graph algorithms based on algorithms for connected components, the Euler tour technique and ear decomposition.

the ACU. In addition to the problem of efficient memory access, the size of the local memory of the ACU is not large enough to put all the global data we need. Instead, we partition the local memory bank of each PE into two halves. One of them, which we call the *global data bank* of each PE, is mapped onto part of the global memory bank and the other half, which is called the *local data bank* of each PE, is used for storing local data for local computations. The entire local memory of the ACU will be part of the global memory of the PRAM model. When implementing a PRAM algorithm on the MasPar architecture, we put information that is most frequently used by a certain RAM into the global data bank of that particular PE. We put common data used by all the PE's into the local memory bank of the ACU and arrange for the ACU to broadcast the needed data to all PE's. We illustrate the mapping in Figure 10.3.

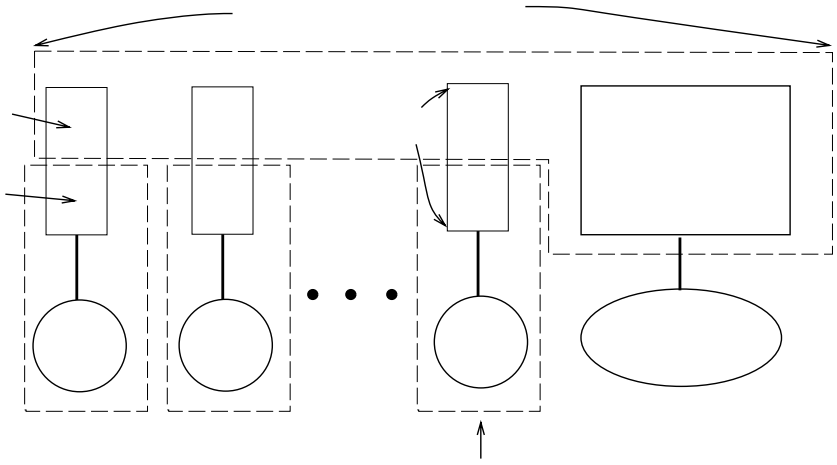


Figure 10.3: Mapping from the MasPar architecture to the PRAM model.

10.5 Overview

The organization for the rest of the chapters in Part II is as follows. In Chapter 11, we present our first implementation of a set of parallel algorithms for basic graph problems on the massively parallel machine MasPar, together with some performance data. We also describe the interface we developed for these algorithms with NETPAD.

The implementation of parallel graph algorithms described in Chapter 11 has the constraint that the maximum input size is restricted to be no more than the number of physical processors on the MasPar. The MasPar language MPL that we used for our code does not support virtual processing. In Chapter 12, we describe techniques for implementing efficient parallel algorithms on the MasPar MP-1 with virtual processing. We first present our data allocation scheme for virtual processing and a set of translating rules for rewriting a code that uses no virtual processors into a code with virtual processing. We then describe the implementation and fine-tuning of a set of commonly used routines with virtual processing. In coding these routines, we

tried different underlying algorithms. We present the performance data for our implementations. We also present the performance data of our sequential code and compare it with the performance data of our parallel code. Our experimental data suggests that by using our techniques, one can implement parallel algorithms with virtual processing efficiently on the MasPar using the MPL language.

Using the techniques described in Chapter 12, we re-coded and fine-tuned our earlier code for parallel graph algorithms (Chapter 11) to incorporate the use of virtual processors. This work is described in Chapter 13. Under this implementation scheme, there is no limit on the number of virtual processors that one can use in the program as long as there is enough main memory to store all the data required during the computation. We also give two general optimization techniques to speed up our computation.

We tested our code with virtual processing on test graphs with various edge densities. We also compared the performance data for our parallel code with the performance data of our sequential code for these problems. We found that the extra overhead for simulating virtual processors is moderate and the performance of our code tracks theoretical predictions quite well, although real-time speed-ups are quite small since the MasPar processors are rather slow. In addition, our parallel code using virtual processing runs on much larger inputs than our sequential code.

Finally, we summerarize our work and list directions for future work in Chapter 14.

We used the following conventions for the names of subroutines provided by the MPL and our implemented routines in Part II. We use the typewriter type style in LaTeX [Lam86] for the names of these MPL subroutines,

e.g., `psort`. The names for the MPL subroutines without virtual processing do not have the prefix letter "v", e.g., `psort`. We add a prefix letter "v" to the name of an MPL subroutine without virtual processing for the name of the same subroutine when virtual processing is allowed, e.g., `vpsort` means an implementation of `psort` with virtual processing.

Chapter 11

Implementation of Parallel Graph Algorithms without Virtual Processing

11.1 Introduction

This chapter summarizes a project we undertook at Bellcore during the summer of 1991 for implementing parallel graph algorithms. In this project, we implemented efficient parallel algorithms for solving several undirected graph problems using the massively parallel computer MasPar [Mas91c] at Bellcore. In addition, we also built an interface between the graph algorithm design package developed at Bellcore called NETPAD [DMM92, Mev91a, Mev91b, MDM91] and our parallel programs. Our purpose was to experiment and set up programming environments for implementing parallel algorithms on massively parallel computers.

The organization of this chapter is as follows. Section 11.2 describes our implementation environment and an interface that we have built between NETPAD and our parallel programs. Section 11.3 describes the implementation details of our parallel graph algorithms. Section 11.4 gives speed-up data of our parallel implementations and an analysis of their performance. Finally, Section 11.5 gives concluding remarks. Work reported in this chapter is based on material presented in [HRD92].

11.2 Implementation Environment

In this section we describe the environment in which we implemented several efficient PRAM graph algorithms. In Section 11.2.1, we discuss the

general issues involved in implementing PRAM algorithms on the MasPar. Our parallel implementations are integrated with a graph algorithm design package called NETPAD [Mev91a, Mev91b, MDM91]. Section 11.2.2 describes the NETPAD software and our interface between the NETPAD and our parallel programs.

11.2.1 Mapping Efficient PRAM Algorithms for Graph Problems

Since the MPL programming language requires that the user takes care of the processor allocation problem, we use the simple strategy of allocating one PE for a node and an edge in our implementation. Hence our implementation can only handle graphs of size less than or equal to the number of PE's in the MasPar. We place data generated for each node or edge into the global data bank of the PE that is in charge of the node or edge. Global variables (for example, the total number of nodes and the total number of edges) are put into the local memory of the ACU. Each PE can access its global data bank efficiently under the mapping; however, global memory accesses to global data banks of other PE's will require going through the global router to get the data. In Chapter 10, we mentioned that a faster way of getting data from the other PE is by going through the XNET configuration. We can use the XNET configuration if PE_i wants to read the global data bank of PE_{i+c}, for all processor elements PE_i, where c is a constant. Some of the global memory accesses required by the PRAM algorithm fall into this category, as in the case when each PE obtains the data from the PE with an ID that is one greater than itself. In our implementation, we always try to take advantage of the XNET configuration whenever possible.

List Ranking on the MasPar One problem that we often face in mapping PRAM graph algorithms onto the MasPar architecture comes from the fact that we usually define the graph using the edge list data structure (an edge list of an undirected graph G is a list of all the edges in G). The PRAM algorithms often link all the edges or all the nodes in a special ordered linked list and perform list processing computations such as list ranking [KR90] (a *list ranking* on a linked list requires each element in the list to compute the suffix sum of all the elements in front of it; the sum could be any associative operator). The list ranking problem on a list of length n can be solved by a sequence of $O(\log n)$ global memory accesses on a PRAM. These global memory accesses can be implemented only as requests to the global router, since elements in a list are not structurally allocated such that we can use the XNET configuration.

An operation on an array of elements called *prefix sums* [LF80, Sch80] (or `scan` [Ble89, Mas91a]) is to compute the prefix sum of all the elements before each array element; the prefix sum operator can be any associative operator. This computation is similar to list ranking, except that the input is in an array rather than a linked list. The prefix sums problem can be solved by a sequence of $O(\log n)$ global memory access on a PRAM. In implementing the PRAM prefix sums algorithm on the MasPar, if we put the ith element of the array into the ith PE, then global memory accesses can be structured in a way that we can make use of the XNET connection. There is already such a routine called `scan` which is implemented in MasPar system library [Mas91a]. The scan operation is very fast compared to the list ranking algorithm we implemented. Note that a linked list can be converted into an array by first performing a list ranking computation and then rearranging the list elements into an array using a global memory write. Since the list ranking operation is one of the most commonly

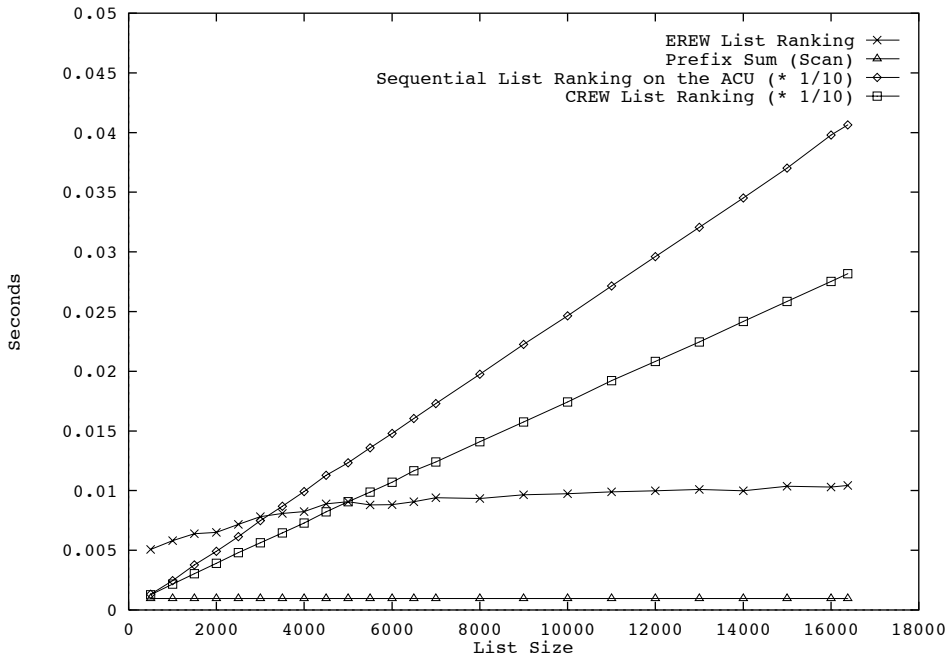


Figure 11.1: Relative performances of sequential list ranking on the ACU, parallel CREW list ranking, parallel EREW list ranking and the prefix sums (scan) operation. In order to fit all data on the same chart, the data for sequential list ranking and CREW list ranking has been divided by 10. The ACU is about 10 times more powerful than each individual PE. Thus on an input list of size 16,384, prefix sums on an array gains a speed-up factor of about 4,000, while EREW list ranking has a speed-up factor of about 380. The prefix sums operation is more than 10 times faster than the EREW list ranking.

used subroutines in PRAM undirected graph algorithms, we convert a linked list to an array if several list ranking operations are to be performed on the list. Figure 11.1 compares the relative speed-up of the list ranking program and the scan (prefix sums) operation.

Concurrent Global Memory Access on the MasPar Some of the PRAM algorithms that we have implemented are based on concurrent read and/or concurrent write PRAM models. If a PRAM global memory concurrent read request is sent to the global router, the global router will automatically satisfy

all concurrent read requests. To understand the behavior of the concurrent read operation executed by the global router, we implemented a routine to transform concurrent read requests to exclusive read requests using the standard simulation algorithm (see, e.g., [KR90]). We found that the performance of our parallel algorithms when using the global router and using our simulated concurrent read routines is very similar.

For concurrent write, the global router allows more than one PE to write into the global memory bank of any particular PE. However, the result of the concurrent write is unpredictable. It does not allow any concurrent write into the local memory bank of the ACU. One can use the system library routine `sendwith` [Mas91a] to implement concurrent write. The `sendwith` routine allows each PE to send data to the global memory bank of any PE. If there is more than one PE trying to send data to any one PE at a given time, all of these data are collected and an associative operator specified by the routine is performed to compute the result, which is then given to the destination PE. For example, a `sendwithMax32`(*item*, *destination*) command tells each active PE to send the 32-bit variable *item* in its own local memory to the PE with the ID equal to its local variable *destination*. If there are several PE's trying to write into any one PE, the result is the maximum value of all variables sent. The `sendwith` operation uses a sorting routine, several non-conflict (EREW) global memory accesses, and a scan operation. The `sendwith` operation can be performed in $O(\log p)$ time on an EREW PRAM model with p processors.

In our implementation of PRIORITY CRCW algorithms, we used the global router to satisfy concurrent read requests and the `sendwith` operations to implement concurrent write requests.

11.2.2 NETPAD Interface

The NETPAD software [Mev91a, Mev91b, MDM91] is a general tool for graph manipulations and graph algorithm design. It uses the X-window system [Nye88], and one can edit graphs and display them easily. The NETPAD system also provides a rich set of basic graph manipulation operations. By calling these operations in one's own program (preferably written in C [KR78]), users can implement their own graph algorithms easily.

NETPAD was used in the following ways to support the design of our parallel graph algorithm packages. It provided routines for generating test graphs. We also used it as a standard interface to input graphs generated by other packages. Most importantly, we used it to display the output of our parallel graph algorithms. Since NETPAD has been designed primarily for supporting sequential computations, it was necessary to build an interface between it and the parallel programs that we implemented on the MasPar. We describe the interface in the following paragraphs. Figure 11.2 gives a schematic diagram of this interface.

For each MasPar routine that we implemented, we wrote a NETPAD external program in C language (with a suffix ".c" in its filename) and included it in one of the NETPAD pop-out menus. While running NETPAD software in the front end, this external program would be invoked through the NETPAD system. The NETPAD system uses the Unix `fork` system call to create another process in the front end to run the external program. The graph in the current window is saved into a file and the name of that file is passed to the forked process. The external program first uses the NETPAD system routines to retrieve the input graph from the input file. The NETPAD system routines are also used to collect the edge list for the graph and store it in an array inside

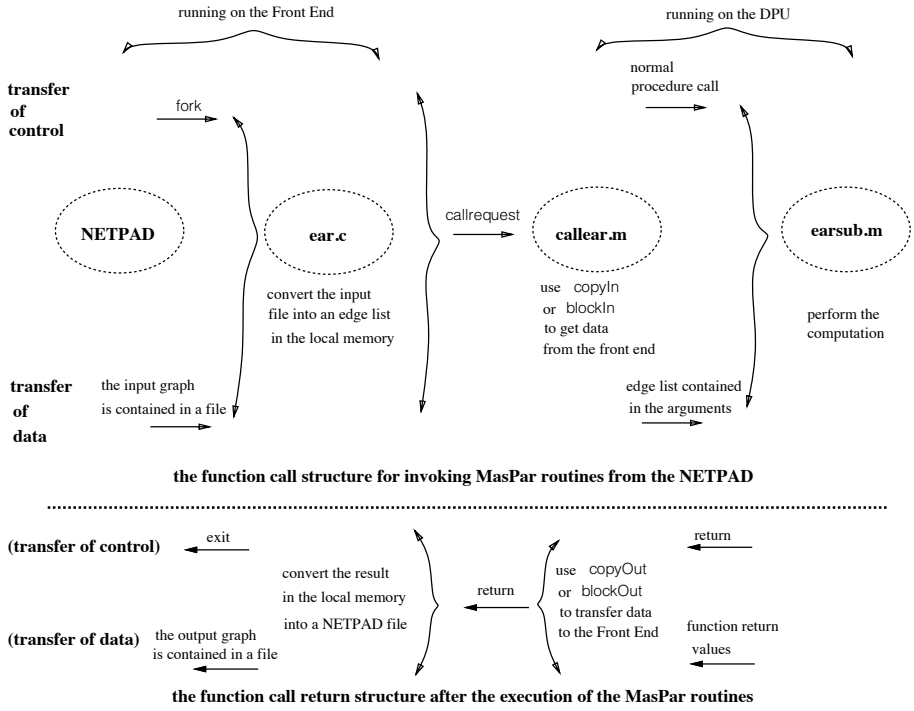


Figure 11.2: Interface for calling the MasPar routines for executing the ear decomposition algorithm from NETPAD.

the local memory bank of the front end. (The external program that we wrote can also be executed independently from the NETPAD system. As long as we have a file that describes the input graph using the NETPAD format, we can run the external program under the Unix system without going through the NETPAD system.)

After the external program collects the edge list, we can request a MasPar MPL program (with a suffix ".m" in its filename) to be executed in the DPU using a special MasPar system call named `callrequest`. An interfacing MPL program must be written for each MasPar routine that we want to use. This routine is invoked by the external program running on the front end by using `callrequest` with arguments describing memory addresses (of the front end) that will be used by the DPU. The MPL program first uses the system

routine `copyIn` to copy the contents of any consecutive block of memory in the front end to the local memory of the ACU. Then it uses the system routine `blockIn` to copy the contents of any consecutive block of memory in the front end to each PE. Routine `blockIn` takes two sets of parameters. The first set describes the starting location of the front end memory to be copied and the size of each memory cell. The second set of parameters describes the rectangular block of PE's that are to receive data. The routine puts the ith cell into the ith PE within the block. The MPL routine then calls the MasPar library that we have implemented to perform the actual computations for the parallel graph algorithms.

After completing the MasPar computations, the MPL interfacing program uses the system routine `copyOut` to copy any data in the local memory of the ACU to the local memory of the front end. A similar routine `blockOut` transfers data from a rectangular block of PE's to a consecutive block of memory locations in the front end. After termination of the MPL interfacing routine, the external program running on the front end takes the output graph in its local memory and writes the output graph into a file using the NETPAD format. The external program exits and sends a signal to the NETPAD system. The NETPAD system gets the output file and displays the graph on the drawing window.

In Figure 11.2, we illustrate this interface by using an example of calling a MasPar routine to perform ear decomposition [Ram93] from NETPAD.

11.3 Our Implementation of Parallel Graph Algorithms

In this section, we describe the parallel graph algorithms we have implemented. In Section 11.3.1, we describe the data structures used in implementing the algorithms. We give the structure of the library of programs

we have implemented, along with a brief description of techniques used for fine-tuning each program in Section 11.3.2.

11.3.1 Data Structures

Array and Linked List We map a global memory array used in a PRAM algorithm onto the MasPar by putting the ith element of the array into the ith PE. We map a linear linked list used in a PRAM algorithm by putting each element in the list into a different PE. Pointers in PRAM are replaced by PE ID's.

Tree We represent an edge in an undirected tree by two directed edges with opposite directions. A tree is represented by a list of directed edges. In implementing the tree data structure on the MasPar, we put one directed edge in one PE with the requirement that the set of edges that are incoming to the same vertex have to be allocated in a segment of PE's with consecutive ID's. Using this representation, we can use the XNET configuration to perform the interprocess communications needed for computing an Euler tour on a tree. Since computing an Euler tour is one of the most commonly used routines for performing tree-based parallel computations, we save time by using this type of mapping.

Undirected Graph A general undirected graph is also represented by a list of edges. Each edge has two copies with the two endpoints interchanged. On the MasPar, we put an edge on a PE with the requirement that the two copies of the edge have to be located in adjacent PE's.

Let the input graph contain n vertices and m edges. We number vertices from 1 to n. Since we have to take care of processor allocation when

using the MPL programming language, we use the following method to allocate processors. The PE with ID equal to i performs computations for the ith vertex in the PRAM algorithm. Although the PE's of the MasPar are numbered from 0, we do not use the PE with the ID 0 for convenience of programming. Each edge has two copies which have consecutive ID's. The PE with ID equal to i performs computations for the ith edge in the PRAM algorithm. Under this scheme, our current implementation can perform computations on graphs with $p-1$ vertices and $\lfloor \frac{p-1}{2} \rfloor$ edges, where p is the number of PE's on the MasPar.

11.3.2 The Parallel Graph Algorithms Library

To build our parallel graph algorithms library, we first wrote a kernel that includes all of the commonly used subroutines for designing parallel graph algorithms. Then we built our graph application programs by calling routines in the kernel. The structure of the whole library is shown in Figure 11.3. We first describe routines in the kernel.

Routines in the Kernel All of these routines are based on PRAM algorithms that run in $O(\log n)$ time using n processors for an input of size n. These are not optimal algorithms, but they are within an $O(\log n)$ factor of optimality, and they are very simple. Our implementations were only for input of size less than 16,384, so the overall optimality was not a serious problem.

1. **List ranking.** We implemented EREW PRAM list ranking routines that compute the rank of each element in a list, where each element in the list is stored in a different PE with a pointer that points to the ID of the PE that stores the next element in the list. These routines require $O(\log n)$ global memory accesses on a list of n elements.

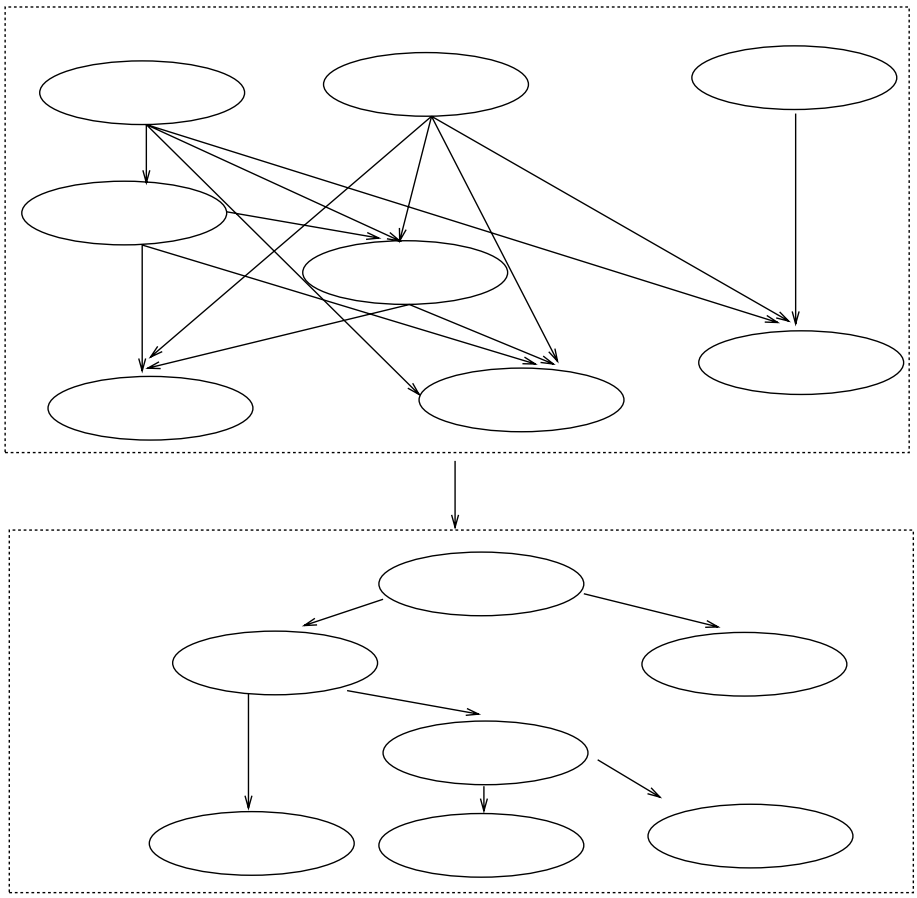


Figure 11.3: The structure of the routines we built for the parallel graph algorithm library: The kernel of the library will be used by the application routines. An arrow from one node to another node means the routine at the tail of the arrow (upper) will be used by the routine at the head of the arrow (lower).

2. **Rotation.** These routines rotate the data stored in the PE with ID i to the PE with ID $(i + d) \bmod p$, where d is a constant and p is the number of PE's in the system. The rotation routines make use of the mesh-connected XNET connection and are faster than implementing the same functions using the global router. These routines require 2 XNET communications.

3. **Segmented rotation.** We store data in each PE and partition PE's into sequences of consecutive segments. These routines rotate the data stored in each PE within each segment. Data within each segment are rotated in a way similar to the rotation routines described in (2). These routines use a constant number of XNET communications and non-conflict (EREW) global memory accesses.

4. **Range minimum.** Let v be a local variable stored in each PE. We build a table such that given a starting ID s_i and an ending ID e_i in each PE, we can compute the minimum value of all v's from PE s_i to PE e_i using only one global memory access. For this, we implemented the $O(\log n)$-time $O(n)$-processor PRAM algorithm in [TV85] for solving the range minimum problem on n elements. In this algorithm, the n elements are stored in an array A. We are required to build a two-dimensional array $W[i, j]$, such that $W[i, j] = \min_{k=0}^{2^j-1} A[i + k]$, for all $0 \leq j \leq \lceil \log n \rceil$ and $1 \leq i \leq n - 2^j + 1$. We construct the two-dimensional array by storing all elements of $\{W[i, j] \mid 0 \leq j \leq \lceil \log n \rceil\}$ in the ith PE. Thus each PE has a one-dimensional local array. Since each global memory access is very structured in the algorithm for building the table, we can use XNET communications to implement global memory accesses used in this algorithm. Let k_i be the least integer such that 2^{k_i} is greater than

or equal to $\frac{e_i - s_i + 1}{2}$. To process each query, a PE with ID i reads $W[s_i, k_i]$ and $W[e_i - 2^{k_i} + 1, k_i]$ through the global router and returns the minimum of the two values.

5. **Euler tour construction** [TV85]. Given a tree represented by an adjacency list, we store each edge of the adjacency list in a different PE. Edges that are adjacent to a vertex are stored in consecutive PE's. By replacing each undirected edge between two vertices with two directed edges of opposite directions, the algorithm returns an Euler tour for the input tree by building a linear linked list. The routine uses the segmented rotation routines, list ranking routines, rotation routines and one global memory access.

6. **Preorder numbering.** Given a tree, we assign a consecutive preorder numbering for its vertices starting from 1. To implement this, we have to use list ranking and a constant number of global memory accesses.

7. **Least common ancestor.** This routine finds the preorder number of the least common ancestor for any given pairs of vertices in a rooted tree. Each PE stores two vertex ID's of the tree. For each PE, the routine returns the vertex which is the least common ancestor of the two vertices in the input rooted tree. The routine uses the range minimum queries and a constant number of global memory accesses.

Graph Application Routines We now describe several graph algorithms that we implemented using the above kernel.

1. **Connected components.** We implemented the CRCW PRAM algorithm described in [AS87]. The PRAM algorithm runs in $O(\log n)$ time

using $O(n+m)$ processors on a graph with n vertices and m edges. The concurrent read operation was implemented by using the global router. The concurrent write operation needed in the algorithm was implemented by the `sendwith` operation. After the execution of the algorithm, the ith PE gets a number indicating the connected component containing the ith vertex. The component number is the least vertex number in the connected component.

2. **Spanning forest.** We modified the algorithm in [AS87] for finding connected components to find a spanning forest of the input graph. The original algorithm partitions the set of vertices into a set of disjoint sets such that vertices in each set are in the same connected component. Initially, the algorithm puts a vertex in each set. As it executes, the algorithm merges two sets of vertices if they are in the same connected component. Our program selects an edge connecting a vertex in one set to a vertex in the other set while merging these two disjoint vertex sets.

3. **Minimum cost spanning forest.** Given an input graph with an integer weight assigned on each edge, we modify the algorithm in [AS87] for finding connected components to find a minimum cost spanning forest for the input graph. This algorithm also partitions the graph into disjoint sets of vertices. In addition, for each current set of vertices, we compute a minimum-cost edge with exactly one endpoint in the set using the `sendwith` operation. This edge determines which other set of vertices is to be merged with its set. Once the merge is completed, the edge that caused the merging is marked as one of the edges in the minimum cost spanning forest.

4. **Ear decomposition of a two-edge-connected undirected graph.** We implemented the PRAM parallel algorithm in [Ram93] for finding an ear decomposition by calling the MasPar system sorting routine, routines in the kernel and the routine for finding a spanning forest.

5. **Open ear decomposition of a biconnected undirected graph.** We implemented the PRAM parallel algorithm in [Ram93] for finding an open ear decomposition by calling the MasPar system sorting routine, routines in the kernel, the routine for finding a spanning forest, and our parallel routine for finding an ear decomposition.

6. **Strong orientation of a two-edge-connected undirected graph.** We first find an ear decomposition for the input graph. Then we direct the edges of each ear so that each ear forms a directed path or a directed cycle. Observe that the ear decomposition algorithm first finds a rooted spanning tree T. The edges in an ear are of the form (v_1, v_2), (v_2, v_3), ... , (v_{k-1}, v_k), (v_k, u_r), (u_r, u_{r-1}), (u_{r-1}, u_{r-2}), ... , (u_2, u_1), where (1) (v_i, v_{i+1}) is a tree edge and v_i is the parent of v_{i+1} in T, for $1 \leq i < k$; (2) (u_{i+1}, u_i) is a tree edge and u_{i+1} is the parent of u_i in T, for $1 < i \leq r$; (3) (v_k, u_r) is a non-tree edge. Thus we direct every non-tree edge (u, v) from u to v where u has a smaller ID than v. Then we assign directions to tree edges in such a way that the edges in an ear form a directed path or directed cycle and the first two ears together form a directed cycle.

7. **Cut edges.** We first find a rooted spanning tree T for the input graph G. (The current version of the program requires G to be connected.) A cut edge is a tree edge (u, v), where u is the parent of v and there is no non-tree edge (x, y) in G such that either x or y is a descendant of v or

equal to v and the least common ancestor of x and y is an ancestor of u or equal to u. This can be implemented by using the Euler tour technique and the range minimum queries.

11.4 Performance Data

In this section, we present performance data for our parallel programs. In Section 11.4.1 we first describe the set of sequential algorithms that we have implemented corresponding to the parallel graph algorithms given in Section 11.3.2. Then in Section 11.4.2, we describe the way we tested and made measurements on the running time of the sequential programs and their parallel counterparts. We obtained their CPU running time and computed the speed-up factor between our parallel programs and sequential programs. This data is presented and analyzed in Section 11.4.4.

11.4.1 Sequential Algorithms

After we implemented the parallel graph algorithms library, we also implemented the following sequential graph algorithms using NETPAD.

1. A recursive version of depth-first search for finding connected components.

2. A routine for finding all cut edges in the graph based on the recursive version of depth-first search [Tar72].

3. A routine for finding a strong orientation based on the recursive version of depth-first search [Tar72].

4. A routine for finding an ear decomposition based on the recursive version of depth-first search [Ram93]. This routine also finds an open ear decomposition on a biconnected graph.

5. Kruskal's algorithm [Tar83] for finding a minimum cost spanning forest.

All but the last of the above five routines are based on depth-first search with some special bookkeepings. The routine for finding an ear decomposition also needs a linear time bucket sort routine. For depth-first search, finding cut edges and finding a strong orientation, the running time is linear in the size of the graph (with a very small constant factor).

The routine for finding an ear decomposition runs in linear time, but has a slightly larger constant factor because of the usage of the bucket sort routine. Kruskal's algorithm for finding a minimum cost spanning forest runs in $O(n+m \log n)$ time on an input graph with n vertices and m edges. Although faster algorithms are known for this problem [Tar83], we implemented Kruskal's algorithm because of its simplicity.

11.4.2 Testing Scheme

We ran our sequential programs a SPARC II workstation with 32 megabytes of main memory. We wrote a random graph generator using NETPAD and generated random graphs for various input sizes.

We tested our programs on graphs with more edges than vertices. To generate a random graph with n vertices and m edges, we first generated an empty graph with n vertices. Then we added one edge at a time where each edge being chosen with uniform probability until all m edges were generated. To generate a connected test graph with n vertices and m edges, we first generated an empty graph with n vertices. Then we constructed a spanning tree by adding edges to connect different connected components with each edge being chosen with uniform probability over all candidate edges. Finally, we randomly added

edges until the graph contained m edges. For testing the algorithm for finding a minimum spanning forest, we assigned a random integer cost ranging from 0 to 999 to each edge in the graph with each number being equally likely to be chosen (with repetition).

We generated a biconnected test graph with n vertices and m edges by first generating an empty graph with n vertices. We then chose a random length k, $3 \leq k \leq n$, and k isolated vertices at random. We randomly permuted these k vertices and constructed a simple cycle by adding an edge between every pair of adjacent vertices in the random permutation and by adding an edge between the first and last vertices in the permutation. After that, we added non-trivial open ears of random lengths to connect all isolated vertices. To add a non-trivial open ear, we chose a random length l, $1 \leq l \leq x$, where x was the number of remaining isolated vertices. We randomly picked two non-isolated vertices u and v (without repetition). We then randomly permuted l isolated vertices and constructed a simple path by adding an edge between every pair of adjacent vertices in the random permutation. We added an edge between u and the first vertex in the above permutation and another edge between v and the last vertex in the above permutation. After connecting all isolated vertices, we randomly added edges until all m edges were generated. A 2-edge-connected test graph with n vertices and m edges was generated in a similar fashion by "growing" ears that could possibly be cycles.

Let n and m be the number of vertices and the number of edges in the input graph, respectively. Given any input size (the number of vertices), we generated graphs with three different densities: sparse graphs with $m = \frac{3}{2}n$; intermediate-density graphs with $m = n^{1.5}$; dense graphs with $m = \frac{n^2}{4}$. For each input size on each density, we generated 4 different random graphs. On

each input graph, each program was run 10 times. (We ran each program 10 times on the same input data in order to even out any fluctuation in the Unix routines we used for measuring CPU running time.) The CPU time used in each run was collected. (We only measured the part of the CPU time used for graph computations. The overhead for input/output and for using the NETPAD system was not included.) We then calculated the average CPU time used by the sequential programs for each input size on each density.

The same set of testing graphs was then fed into our parallel programs. On each input graph, we ran the parallel program 10 times. The CPU time used for graph computation was collected for each run. We then calculated the average CPU time used by the parallel programs for each input size on each density.

11.4.3 Least-Squares Curve Fitting

We applied the least-squares fit package in Mathematica [Wol88] to the data we obtained. We used the following method to find the fitted curves for our performance data. We first derived the theoretical asymptotical running time for our parallel program. For example, our code for finding a spanning forest in a graph with n nodes and m edges runs in $O(\log^3 n)$ time since an $O(\log^2 n)$ time sorting routine is needed to implement global concurrent write operations. We first used Mathematica to find coefficients c_0, c_1, c_2 and c_3 such that the function $c_0 + c_1 \cdot \log x + c_2 \cdot \log^2 x + c_3 \cdot \log^3 x$ best fit the set of experimental data that we obtained.

If any of the coefficients was negative, we forced the negative coefficient c_i with the largest integer i to be zero and perform the fitting once again. We iterated this process until all coefficients were not negative. We also performed the least-squares fit for performance data of the sequential programs

when the amount of memory used in the program was within the capacity of the main memory.

To test the goodness of the curve we obtained, we computed the *average error* as the square root of $\frac{1}{k} \cdot \sum_{i=1}^{k} (\frac{y_i - f(x_i)}{f(x_i)})^2$, where k is the number of experimental data points, f is the function that describes the fitted curve and y_i is the experimental value when the input size is x_i.

11.4.4 Analysis

For each graph problem that we solved sequentially and in parallel, we have plotted the relative CPU time used by both programs on sparse graphs, intermediate-density graphs and dense graphs. The results are shown in Figure 11.4 for finding connected components; Figure 11.6 for finding a minimum cost spanning forest; Figure 11.8 for finding an ear decomposition of a 2-edge-connected graph; Figure 11.12 for finding all cut edges; Figure 11.10 for finding a strong orientation of a 2-edge-connected graph; Figure 11.14 for finding an open ear decomposition of a biconnected graph. Although each PE is much slower than the SPARC workstation, we found that in the case of finding a minimum cost spanning forest and finding an ear decomposition, parallel programs in fact run faster in real time compared to sequential programs. For example, the routine for finding an ear decomposition on the MasPar is about 3 times faster (in real CPU time) on the largest test graph we have than the one that runs on the SPARC II workstation.

To compute the relative speed-up between our parallel programs running on the MasPar and the sequential programs running on SPARC workstations, we need to have the ratio of the computation speed of a MasPar PE to that of a SPARC workstation. In [Mas91c] it is stated that each PE is about

10 times slower than the MasPar ACU. We tested programs running on the MasPar ACU and the MasPar front end. Test data show the current MasPar front end, which is a micro-VAX workstation, is about 2 to 3 times faster than the MasPar ACU. We then ran our sequential programs on the MasPar front end. Test data show that the front end is at least 10 times slower than the same programs running on the SPARC II workstation. In some tests, it is more than 15 times slower. (This figure is confirmed by data in [IEE91].) Thus the SPARC II workstation is at least 200 times faster than each MasPar PE.

We rescaled the CPU time used by sequential programs running on the SPARC II. according to the above figures and computed the speed-up of the parallel programs running on the MasPar relative to the sequential programs on the SPARC II. All of our parallel programs run in $O(\log^3 m)$ time using $2m$ processors on an input graph of m edges. Thus the theoretical speed-up for our parallel programs using p PE's is $\Theta(\frac{p}{\log^3 p})$ for all problems, except for the one for finding a minimum cost spanning forest. The theoretical speed-up for finding a minimum cost spanning forest is $\Theta(\frac{p}{\log^2 p})$ using p PE's since the sequential algorithm runs in $O(n + m \log n)$ time. We plotted the relative speed-up for each problem with its theoretical speed-up curve. In plotting each theoretical speed-up curve, we used a constant multiplicative factor that best approximated the experimental data. The results are shown in Figure 11.5 for finding connected components, in Figure 11.7 for finding a minimum cost spanning forest, in Figure 11.9 for finding an ear decomposition of a 2-edge-connected graph, in Figure 11.13 for finding all cut edges, in Figure 11.11 for finding a strong orientation od a 2-edge-connected graph, and in Figure 11.15 for finding an open ear decomposition of a biconnected graph.

We note that the average error for fitted curves on dense graphs and intermediate-density graphs is almost the same whether we fit the data with

functions dominated by $\log^3 x$ or $\log^2 x$, though we used functions dominated by $\log^3 x$ in this chapter. However, the average error for sparse graphs is about twice as large if we use functions dominated by $\log^2 x$ instead of $\log^3 x$.

In terms of actually writing code, we wrote about 4,000 lines of MPL code for our library of parallel programs which includes the kernel and the graph algorithms. We used about 1,600 lines of C code for our sequential programs with the help of NETPAD library routines. Without the help of NETPAD to take care of the general graph data structures and other commonly used graph operations, the size of our sequential programs would have been larger. (In our parallel programs, we only use NETPAD for input and output.) Thus the code for our parallel algorithms was not much larger than that for the sequential algorithms.

11.5 Concluding Remarks

We have implemented several efficient PRAM graph algorithms on the MasPar. We have developed an interface between the graph manipulation package NETPAD and our parallel programs. We have also written sequential programs for solving the same graph problems and studied the relative speed-up of the parallel programs over the sequential ones. Although our implemented parallel code is not currently cost-effective as compared to the performance of our sequential code, our parallel code looks promising when cheaper parallel machines are available in the future due to market demands as one would expect the speed of a sequential machine will eventually reach its limit.

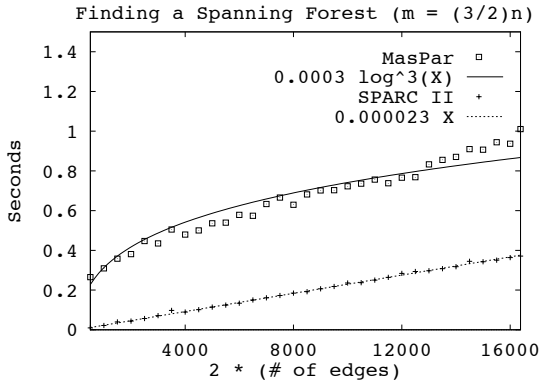
We note a few observations.

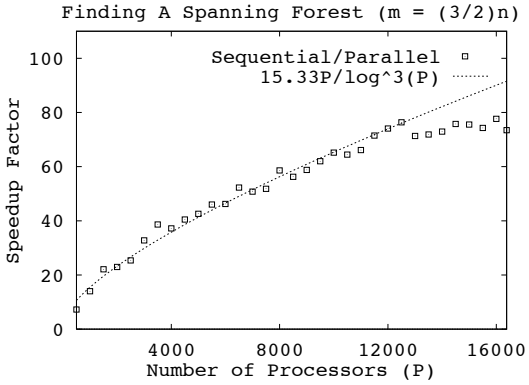
- **PRAM-based graph algorithms can be implemented efficiently and easily.** The PRAM model has proven to be a very good theoretical

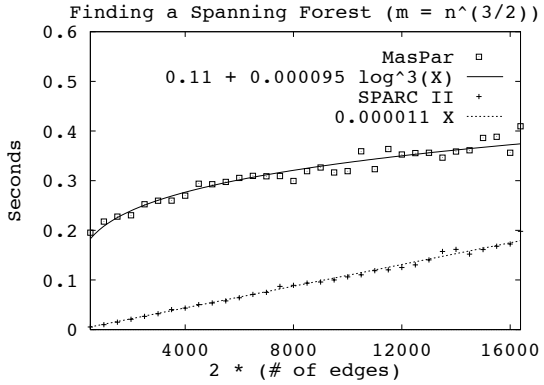
model for designing parallel algorithms. By developing a general mapping strategy between the PRAM model and the target machine hardware architecture, we can make use of results developed on the PRAM model. Our experience with the MasPar shows that we can achieve reasonable speed-up by this approach. The whole process of programming and debugging is easy and fast. (All of the work reported here was accomplished within a period of 12 weeks.)

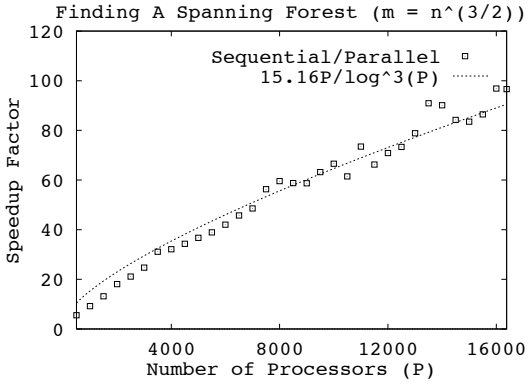
- **Global routing bottleneck.** The current global router on the MasPar is very slow compared to the XNET configuration. (It is 100 times slower than the XNET for transferring a 32-bit data to each PE [Mas91b].) Although we use the XNET configuration when we can in our implementations, our parallel graph algorithms often need to use the global router for performing list computations. The performance of our parallel programs would be significantly improved if the global router could be made to run faster when routing large data sets.

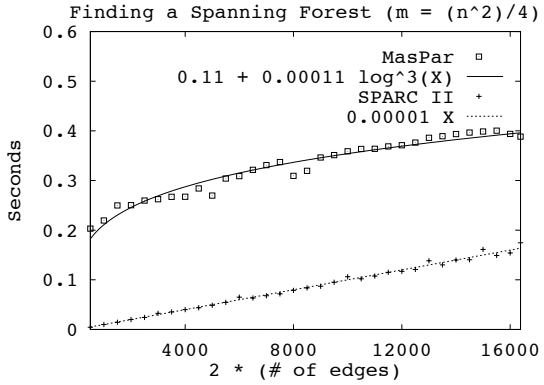
- **NETPAD is a useful tool for designing graph algorithms.** NETPAD takes care of the input of test graphs and the output of results. NETPAD also provides an interface for generating test data from other programs. Its graphic display capability provides a good tool for debugging. In the design of the sequential programs, NETPAD also provides library routines for maintaining graph data structures. Our parallel programs used NETPAD for the input of test graphs and the output of results.











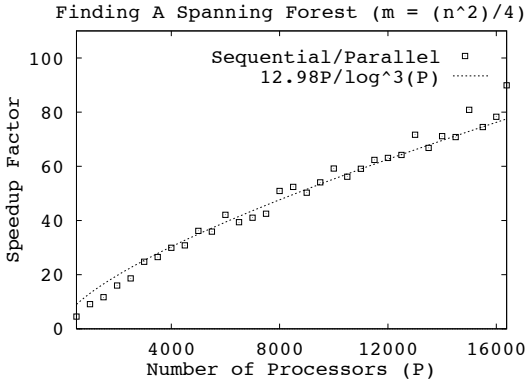
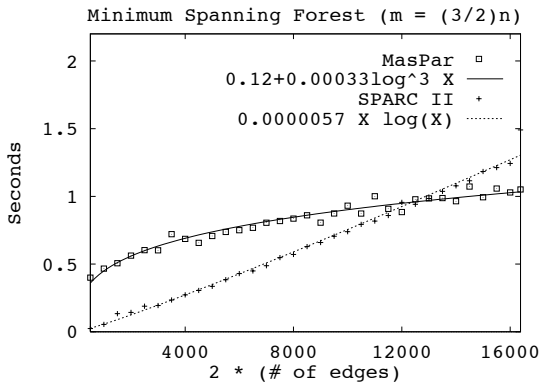
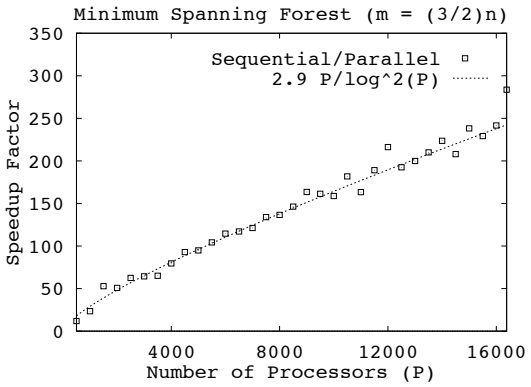
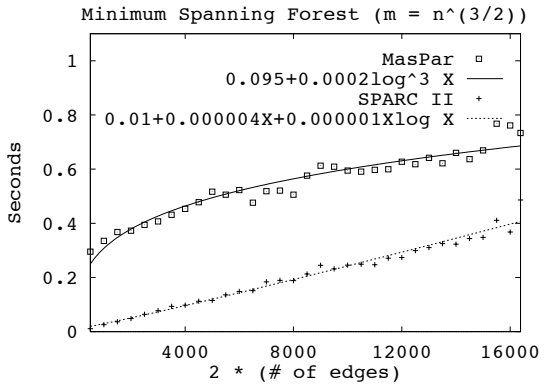


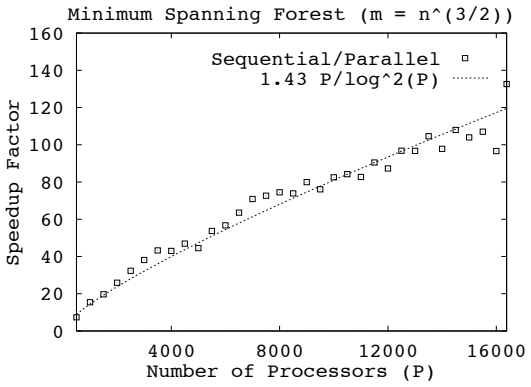
Figure 11.4: Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding connected components.

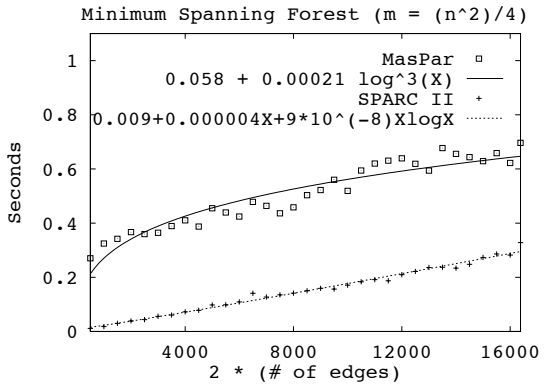
Figure 11.5: Speed-up data for finding connected components.











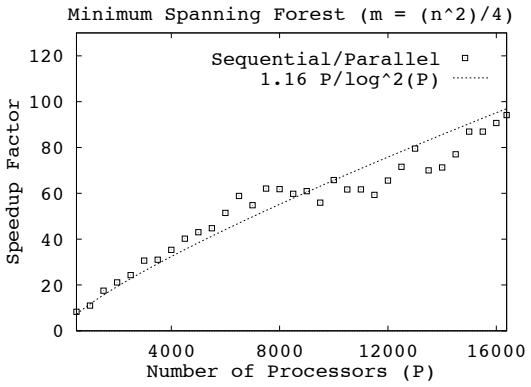
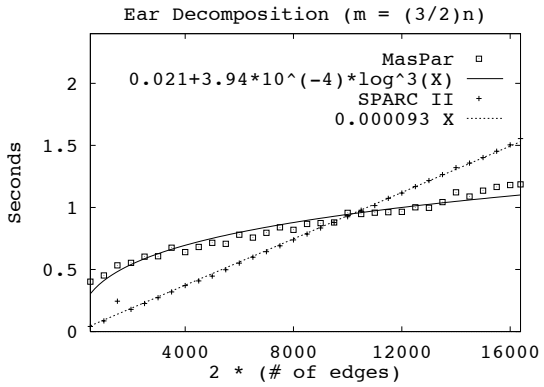
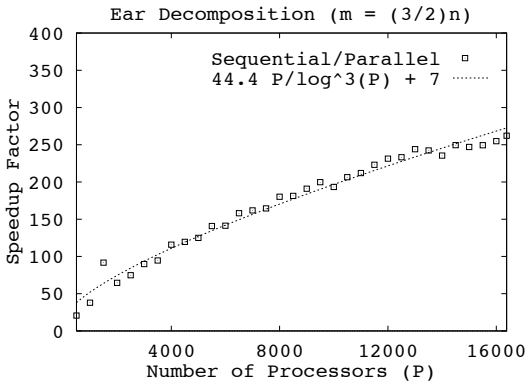
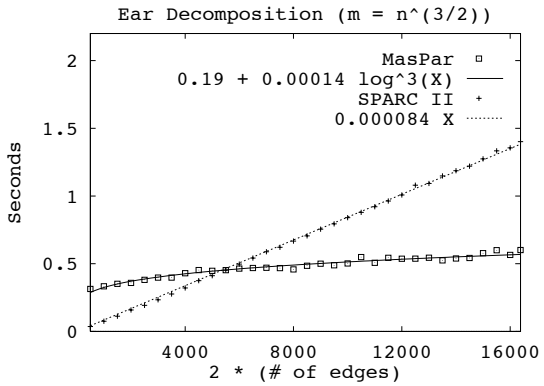


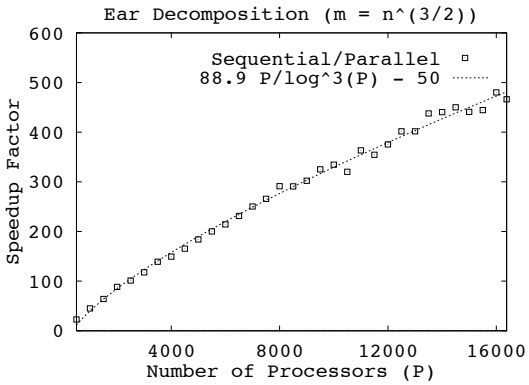
Figure 11.6: Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding a minimum spanning forest.

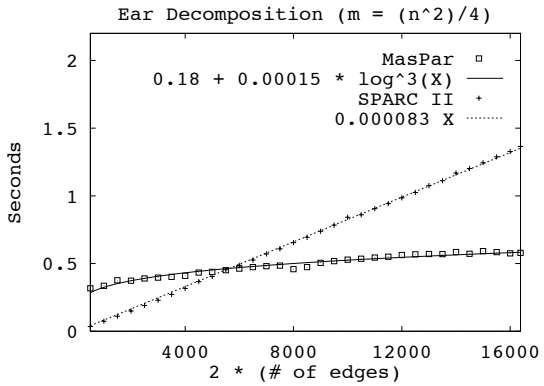
Figure 11.7: Speed-up data for finding a minimum spanning forest.











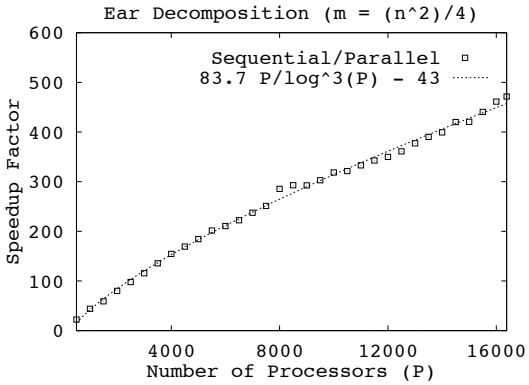
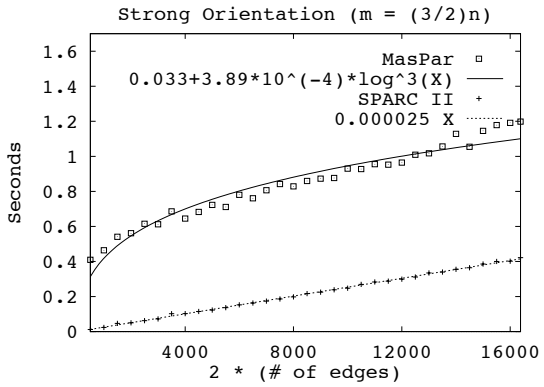
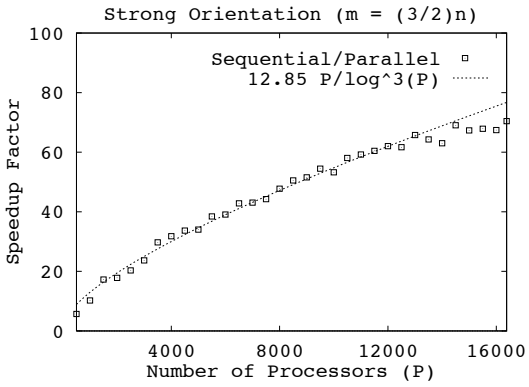
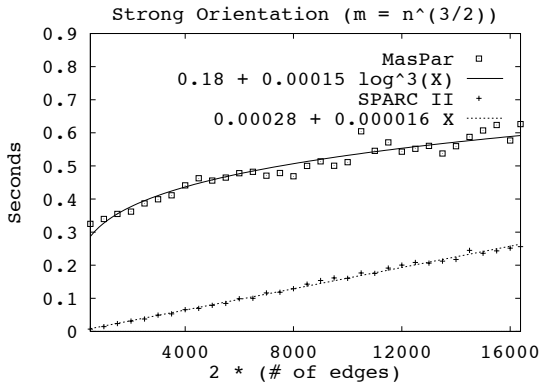


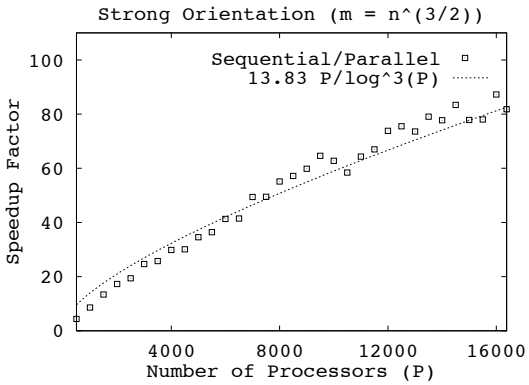
Figure 11.8: Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding an ear decomposition of a two-edge-connected graph.

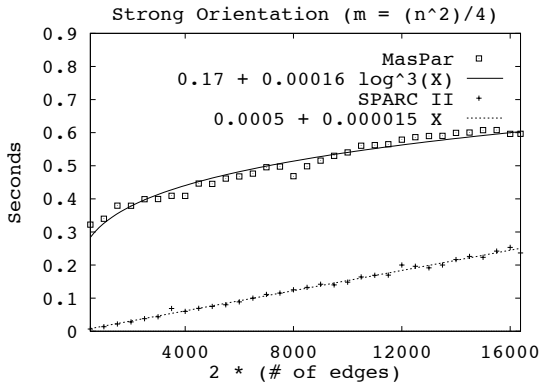
Figure 11.9: Speed-up data for finding an ear decomposition of a two-edge connected graph.











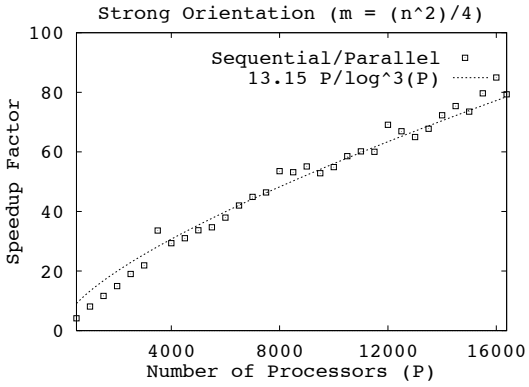
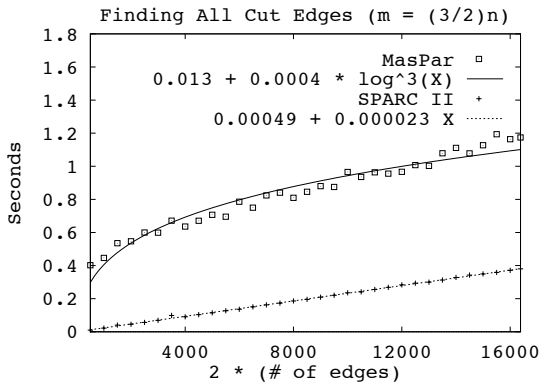
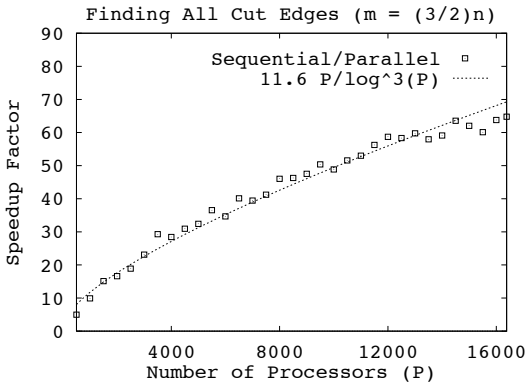
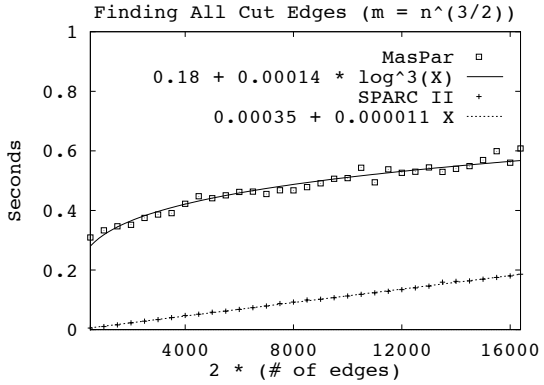


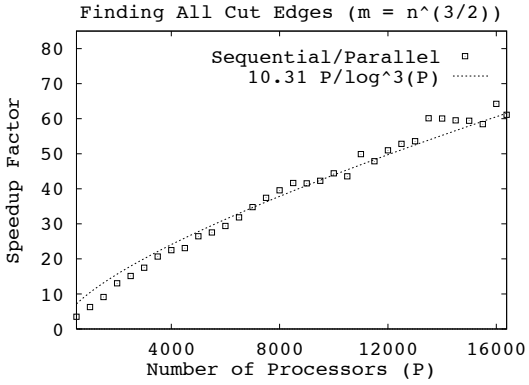
Figure 11.10: Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding a strong orientation of a two-edge-connected graph.

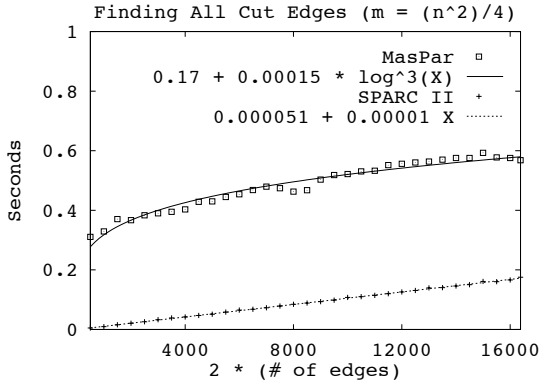
Figure 11.11: Speed-up data for finding a strong orientation of a two-edge connected graph.











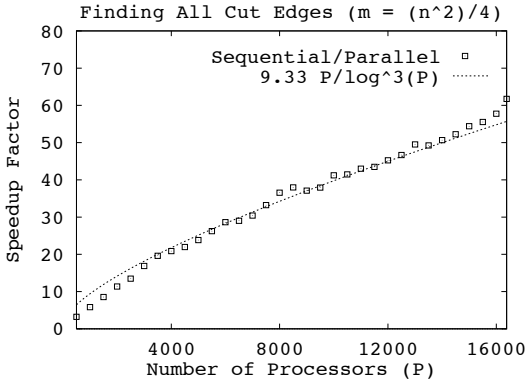
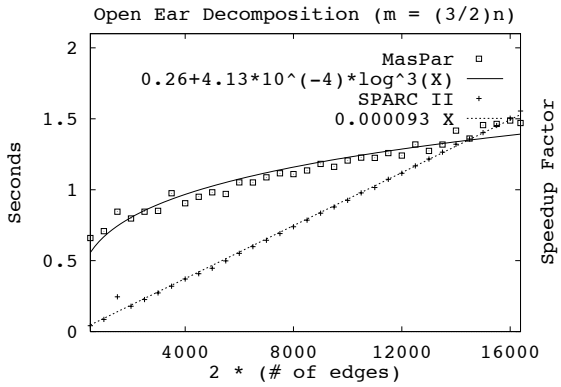
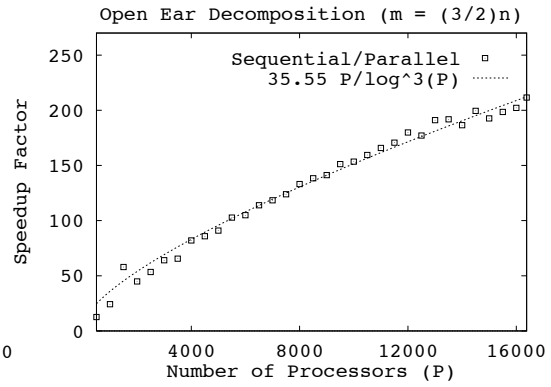
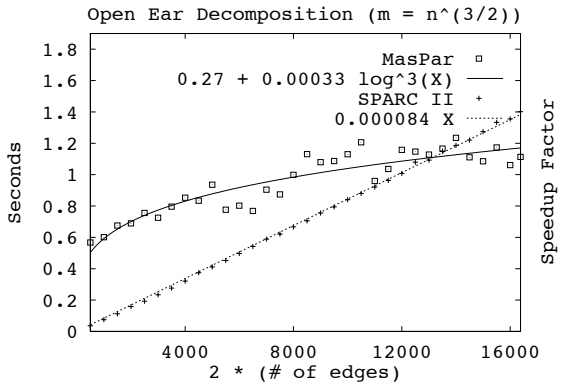


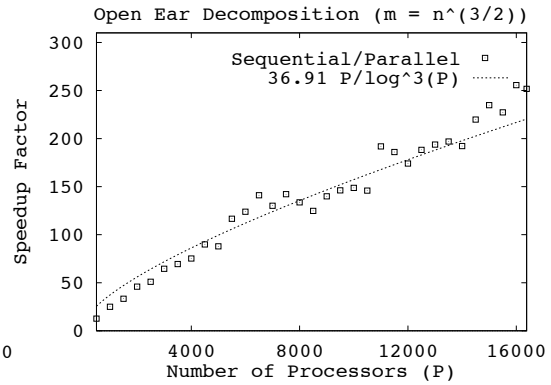
Figure 11.12: Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding all cut edges.

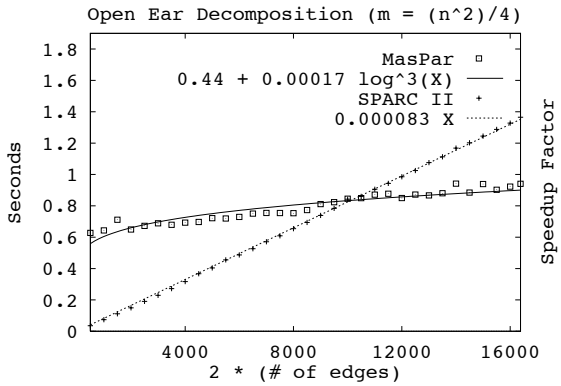
Figure 11.13: Speed-up data for finding all cut edges.











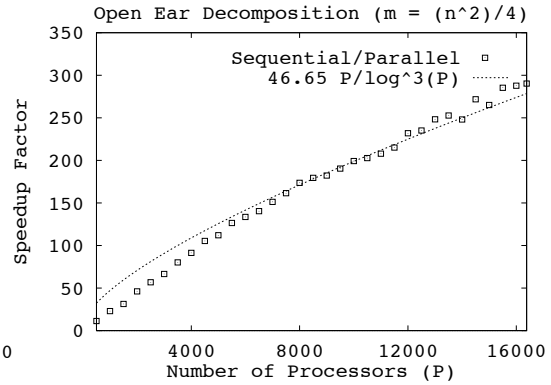


Figure 11.14: Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding an open ear decomposition of a biconnected graph.

Figure 11.15: Speed-up data for finding an open ear decomposition of a biconnected graph.

Chapter 12

Efficient Implementation of Virtual Processing on a Massively Parallel SIMD Computer

12.1 Introduction

This chapter continues the discussion of our implementation project on the massively parallel computer MasPar MP-1. In Chapter 11, we reported the implementation of several parallel graph algorithms on the MasPar MP-1 using the parallel language MPL [Mas92b, Mas92c], which is an extension of the C language [KR88]. (An introduction to MPL is given in [PC93].) The MPL language is a very efficient tool for using the MasPar, but it requires the specification of the physical organization of the processors used in the program. As a result, our implementation described in Chapter 11 could only handle the case where the input size is are no more than the number of available physical processors.

A major advantage in using massively parallel machines with virtual processing is that we can solve problems involving large inputs that cannot be handled by conventional sequential machines. Though the machine that we used (the MasPar MP-1) has only 16,384 processors, we extend our code such that it can handle inputs of sizes larger than 16,384.

Several parallel machines offer the convenience of using virtual processors in their high-level programming languages. For example, the Connection Machine [LAD+92] offers the support of using virtual processors with the

assistance of the hardware and microcode underlying the C* programming language. The parallel FORTRAN language used in the MasPar also supports the usage of virtual processors. However, these supports for using virtual processors come with the penalty of having a very large overhead. Programs that want to achieve a high percentage of machine utilization are either coded in a low-level programming language that supports virtual processing using macros and low-level system routines (e.g., the Paris language in the Connection Machine [BLM+91]) or coded in a language that does not support virtual processing (e.g., the MPL language in the MasPar (Chapter 11 and [PS90])). In the current chapter, we describe techniques for efficiently implementing virtual processing using the latter approach.

Our results are reported in the following sections which are organized as follows. Section 12.2 gives a high-level description of our implementation, and the strategy we used in mapping virtual processors onto the MP-1. Section 12.3 gives a set of rules for rewriting a non-virtual processing code to a code with virtual processing. Section 12.4 describes the implementation and fine-tuning of basic parallel primitives used for parallel algorithms. Finally, Section 12.6 gives the conclusion.

In this chapter, we use *non-virtual processing routines* to denote routines that cannot handle virtual processing.

12.2 Preliminary

12.2.1 High-Level Description of Our Implementation

We use the following techniques for implementing parallel algorithms with virtual processing using the MPL language. First, we implemented a parallel algorithm using the MPL language assuming the size of the input is

no more than the number of available physical processors. We then defined a virtual machine according to the number of virtual processors that we needed. We mapped this virtual machine into the real machine by evenly distributing data allocated to virtual processors among available physical processors. Under this scheme (shown in Section 12.2.2), each physical processor simulated an equal number of virtual processors as follows. Let *vnproc* be the total number of virtual processors requested and let *nproc* be the number of available physical processors. We allocate $\lceil \frac{vnproc}{nproc} \rceil$ virtual processors to physical processor and make the last $nproc \cdot \lceil \frac{vnproc}{nproc} \rceil - vnproc$ virtual processors inactive throughout the computation.

After properly allocating data, we translated the original non-virtual processing code line by line by using rewriting rules given in Section 12.2.2 and Section 12.3 to handle virtual processing. Requests for performing arithmetic and logic operations on data allocated among virtual processors were simulated by several requests performed on physical processors that simulated them. Conditional branching statements were also transformed using a similar strategy. We also provided special mechanism for maintaining the set of active virtual processors throughout the code.

After translating our code using the rewriting rules, we replaced the system routines used in the code. (The MasPar system provides these non-virtual processing system routines for use in the MPL.) We implemented and fine-tuned these system routines and a set of commonly used routines with virtual processing. In the implementation process, we used various techniques and approaches to optimize our code. By replacing the commonly used routines in the original non-virtual processing code with the new primitives with virtual processing, we have implemented the original parallel algorithm with virtual processing.

12.2.2 Mapping of the Virtual Processors onto the MasPar Architecture

In our programs, each virtual processor (or VPE) is given a unique ID ranging from 0 to $vnproc-1$, where $vnproc$ is the number of virtual processors. (Note that $nproc$ is the number of physical processors and they are organized as an $nxproc \times nyproc$ mesh. For the machine that we used, $nproc$ = 16,384 and $nxproc = nyproc = 128$.) The number of virtual processors per physical processor is $vpr = \left\lceil \frac{vnproc}{nproc} \right\rceil$. The virtual processors are arranged into a 2-dimensional $vnxproc \times vnyproc$ mesh.

For our implementation, we used the so-called *hierarchical partitioning scheme* [Mas92a]. Each physical processor simulated a $vpr \times 1$ sub-mesh of virtual processors. Thus given an $nxproc \times nyproc$ 2-dimensional mesh, the virtual machine being simulated is an $(nxproc \cdot vpr) \times nyproc$ 2-dimensional mesh. (The implementation of bitonic sort with virtual processing [PS90] used the same mapping scheme as ours.) We illustrate the mapping in Figure 12.1. The reason for our choice is that in our implementation of parallel algorithms, we frequently need to use operations that can utilize the locality of data (e.g., the prefix sum (scan) operator [Ble89]). This type of data partitioning enables us to preserve the locality of data.

Once our code decided on the vpr value as described earlier in this section, we transform the plural variables used in a non-virtual processing code using the following rewriting rule.

Rewriting Rule 1

(1) Each plural variable allocated in a non-virtual processing code is transformed into a plural array of vpr elements in our new code with virtual processing. The

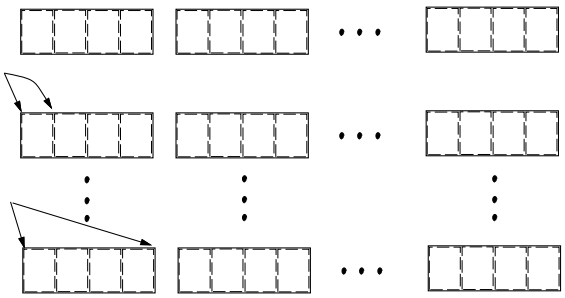


Figure 12.1: Mapping of 4 virtual processors onto each PE.

```
/* without virtual processing */
plural int a;
int b;
```

12.1.(a)

$\Longrightarrow$

```
/* with virtual processing */
plural int a[vpr];
int b;
plural char active[vpr];
```

12.1.(b)

Program 12.1: MPL code segments for data allocation. The variable a is plural and the variable b is singular. A flag *active* is allocated to each virtual processor in our code.

ith element in the jth physical processor corresponds to the local copy of virtual processor $(j-1)\cdot vpr + i$. Variables used in an MPL non-virtual processing code without the plural attribute are not changed in the new code.
(2) An extra flag (called active) in each virtual processor is allocated in the new code to indicate whether its corresponding virtual processor is active during each step of computation.

A small piece of code segment is shown in Program 12.1 to demonstrate our data allocation scheme. Program 12.1.(a) shows the original non-virtual processing code. Program 12.1.(b) shows the transformed code with virtual processing.

Thus given a plural variable *data* and a VPE with ID w, the local copy of *data* is stored in the (w mod vpr)th element of the local array *data* in the PE with ID $\left\lfloor \frac{w}{vpr} \right\rfloor$.

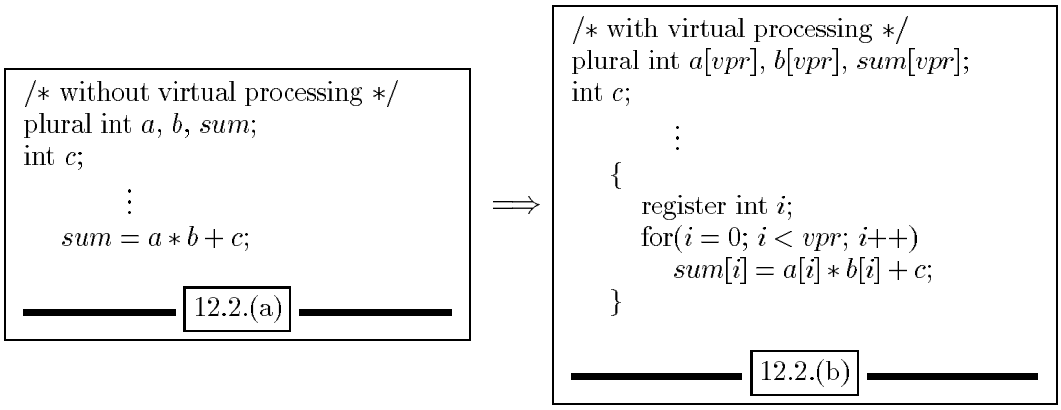
12.3 General Coding Issues for Virtual Processing

The MPL programming language provides two ways of performing SIMD parallel computations. First, the MPL language provides basic arithmetic and logic operations among plural data, and conditional branching statements for performing parallel computations without virtual processing. We discuss the rules for rewriting these statements to handle virtual processing in the following subsections. Second, the MPL also provides subroutines for the fundamental parallel primitives such as computing the prefix sums of an array, sorting and inter-processor communicating when no virtual processor is used. We had to write our own routines for performing the same tasks with virtual processing. We will discuss how to incorporate virtual processing into the system provided subroutines and the commonly used routines given in Chapter 11 in Section 12.4.

12.3.1 Arithmetic and Logic Operations

It was easy to translate arithmetic operations involving the usage of plural variables to handle virtual processing. For each such statement, we transformed the request for computations performed on virtual processors to computations on physical processors. We used the following rule to rewrite statements involved with only arithmetic and logic operations.

Rewriting Rule 2 *Each MPL statement that involves only arithmetic and logic operations can be simulated with virtual processing by vpr sub-steps using available physical processors. In sub-step i, physical processor PE_j, for all j, computes the value for virtual processor $VPE_{(j-1)\cdot vpr+(i-1)}$. The transformed code puts a* **for** *loop around the original statement and converts the plural variables into their array formats as described in (1) of Rewriting Rule 1*



Program 12.2: MPL code segments for arithmetic and logic operations.

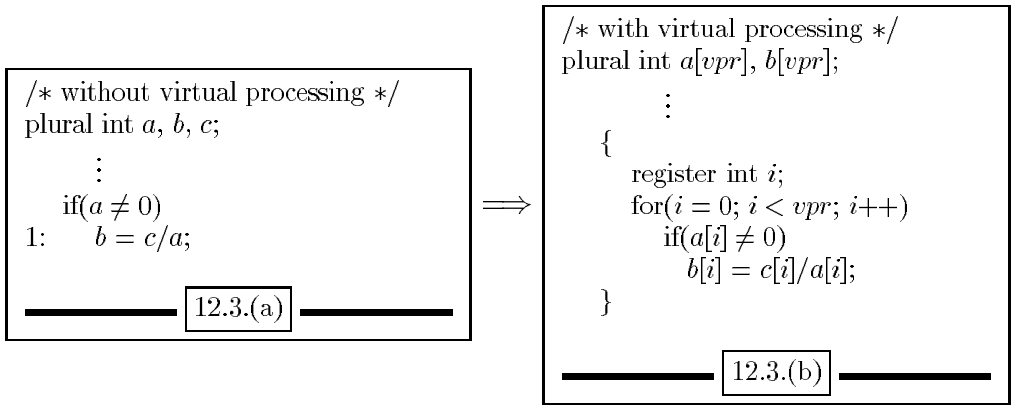
(Section 12.2.2). A group of statements can be grouped into a single **for** *loop as long as each of them involves only arithmetic and logic operations.*

The code rewritten for virtual processing was about twice as long as its non-virtual processing code.

An example is shown in Program 12.2. In Program 12.2.(a), each processor had its own local variables a, b, and sum. The variable c was a global variable stored in the ACU. After performing the statement in Program 12.2.(a), each processor assigned the value of the multiplication of its own a and b plus the value of the global variable c into the local variable sum. The code rewritten to run on a system with virtual processing is shown in Program 12.2.(b).

12.3.2 Conditional Branching Statements

By using an **if** statement to test a local value in each processor, we could decide whether a processor was active or not during each step of computation. We used the following rewriting rule to translate conditional branching statements.



Program 12.3: MPL code segments for conditional branching statements.

Rewriting Rule 3 *An MPL conditional branching statement consists of a conditional test on the value of an expression η, a list of statements that are executed if the value of η is true and an optional list of statements that are executed otherwise. If η consists of only arithmetic and logic operations, we rewrite the conditional branching statement by putting a* **for** *loop around the statement and then converting every plural variable in η into its array format as described in (1) of Rewriting Rule 1 (Section 12.2.2).*

For example, in Program 12.3.(a) each processor with its local value of a being non-zero assigned the value c/a to its local variable b. The local value of b in each processor whose a is zero would not be changed. Thus all PE's with non-zero a values were active in step 1. The rewritten code with virtual processing is shown in Program 12.3.(b).

12.3.3 Procedure Calls

Another way of using an **if** statement for the selection of active processors involves the calling of subroutines using processors that are currently active. We used the following rewriting rule to translate a procedure call statement within an **if** statement.

Rewriting Rule 4 *If procedure calls are used in an expression η that decides the execution of an* **if** *statement, then each procedure call should be evaluated before the* **if** *statement. The results of the procedure calls are saved in temporary variables which are then used to evaluate η. If procedure calls are used inside the body of an* **if** *statement, we use the active flag, defined in (2) of Rewriting Rule 1 (Section 12.2.2), to store the information of which processors are active. The active flag is then passed into the called subroutine. The information about the current set of active processors is saved before the calling of a subroutine and is restored after the subroutine call.*

We show an example in Program 12.4. In Program 12.4.(a), the MPL library routine `reduceAdd32`(a) returned the summation of all a values in active processors. After the execution of the code in Program 12.4.(a), b contained the summation of all local values of a's that were positive. The rewritten code with virtual processing is shown in Program 12.4.(b).

12.4 Implementation of Parallel Primitives

In this section, we describe our implementation of a set of library subroutines with virtual processing. The library routines include several parallel primitives provided by the system and several parallel primitives in the graph algorithms kernel that were given in Chapter 11. None of them could handle the usage of virtual processors.

We classify the parallel primitives into three different categories according to the methods we used to implement them with virtual processing. They will be discussed in detail in the following subsections. Every routine in each category was fine-tuned for speed and small memory space usage. For

```
/* without virtual processing */
plural int a;
int b;
      ⋮
   if(a > 0)
      b = reduceAdd32(a);
```

12.4.(a)

⟹

```
/* with virtual processing */
plural int a[vpr];
int b[vpr];
plural char active[vpr];
          ⋮
   code for saving the current
   "active" values (omitted)
   {
      register int i;
      for(i = 0; i < vpr; i++){
         active[i] = 0;
         if(a[i] > 0) active[i] = 1;
      }
   }
   b = vreduceAdd32(a, active);
   code for retrieving the previous
   "active" values (omitted)
```

12.4.(b)

Program 12.4: MPL code segments for calling a subroutine using a set of selected active processors.

some of the routines, we tried different implementations, and performance data for all of them are reported. We often found that there is a trade-off between the running time and the amount of additional memory used. One can choose the best implementation to suit one's needs. Performance data for routines with and without virtual processing are presented to show the overhead for our implementation of virtual processing. For each type of the operators in each category, we show as an example a representative routine that was implemented. One can implement similar routines in the same type of the operator by making minor changes to our sample code.

12.4.1 Category 1

Parallel primitives in this category could be implemented with virtual processing by exactly one execution of their corresponding original non-virtual

processing codes and some local computations performed in each physical processor. The following parallel primitives are in this category.

- The *reduce* operator which applies an associative operator on a plural variable and outputs its result. For example, the MPL library routine `reduceAdd32` returns the summation of 32-bit integers in each active processor. This routine has been provided in the system library without virtual processing.

- The *scan* operator which applies prefix summing using a given associative operator on a plural variable. For example, the MPL library routine `scanAdd32` computes all of the prefix sums of an array of 32-bit integers for active processors. This routine has been provided in the system library without virtual processing.

- The *neighbor* operator which retrieves data from the processor whose ID is adjacent to the ID of the current processor. This operator is described in the kernel routines given in Chapter 11. For example, the kernel routine `Lneighbor32` retrieves a 32-bit integer from processor $i-1$ for each processor i, $i > 0$. We implemented this operator without virtual processing in the work described in Chapter 11 by using two MPL system mesh-communication operations (i.e., XNET).

- The *segmented rotation* operator which rotates data within a set of processors with continuous ID's. A set of processors with continuous ID's is called a *segment* of processors. Let the next processor to the right of the processor with ID i in a segment of processors be the processor whose ID is $i+1$ if i is not the largest ID in the segment; otherwise its next processor

```
/* return the summation of a values in all active VPE's */
int vreduceAdd32(a, active)
plural int a[vpr];
plural char active[vpr];
{
    plural int tsum;
    register int i;
    /* compute the partial sum in each physical PE */
    tsum = 0;
    for(i = 0; i < vpr; i++)
        if(active[i]) tsum = tsum + a[i];
    /* compute the summation of all partial sums */
    return(reduceAdd32(tsum));
}
```

Program 12.5: MPL Code for adding an array of 32-bit integers with virtual processing.

to the right is the processor in the segment with the least ID. This operator is described in the kernel routines given in Chapter 11. For example, the kernel routine `segRshift32` rotates a 32-bit integer in each active processor to the processor on the right in the same segment. We implemented this operator without virtual processing in the work described in Chapter 11 by using several MPL mesh-communication operations and global routings (i.e., `router`).

As an example, we show the code of `vreduceAdd32` for implementing the MPL system routine `reduceAdd32` with virtual processing in Program 12.5. We could use the naive algorithm of applying the MPL system routine `reduceAdd32` (without virtual processing) *vpr* times and returning the summation of the *vpr* results using the ACU for computing the summation of $vpr \cdot nproc$ elements. However in Program 12.5, we used the MPL system routine `reduceAdd32` exactly once plus performing $O(vpr)$ local arithmetic operations in each physical processor to achieve the same goal. The computation of the MPL system routine `reduceAdd32` requires the use of inter-processor communi-

	Lneighbor32		reduceAdd32		segRshift32		scanAdd32	
	time (ms)	ratio	time (ms)	ratio	time (ms)	ratio	time (ms)	ratio
use no VPE	0.017		0.068		0.747		0.934	
$vpr =$ 1	0.034	2.0	0.116	1.7	1.063	1.4	1.079	1.2
2	0.035	2.1	0.153	2.3	1.146	1.5	1.145	1.2
4	0.059	3.5	0.227	3.3	1.379	1.8	1.285	1.4
8	0.108	6.4	0.375	5.5	1.881	2.5	1.560	1.7
16	0.204	12.0	0.671	9.8	2.882	3.9	2.122	2.3
32	0.398	23.4	1.264	18.6	5.163	6.9	3.235	3.5
64	0.785	46.2	2.449	36.0	9.777	13.1	5.454	5.8
128	1.558	91.6	4.836	71.1	19.087	25.6	9.901	10.6

Table 12.1: Performance data for performing four basic parallel primitives on 32-bit integers. The "ratio" columns give the ratio of the time used by the routines using *vpr* virtual processors per physical processors to the time used by the non-virtual processing routines.

cation, which is much slower than local computation [Mas91b, Pre93b]. Hence we could expect the computation time for `vreduceAdd32` to be faster than the naive algorithm of applying the MPL system routine `reduceAdd32` *vpr* times. Note that the *active* flag, which was defined in (2) of Rewriting Rule 1 (Section 12.2.2), was used in Program 12.5 to mark the set of virtual processors that were currently active.

We list the performance data for our parallel implementation of the algorithms for the four representative examples in this category in Table 12.1. The running time for operators in this category could be formulated as follows. Let $f_p(\mathcal{Q}, vpr)$ be the time to run the algorithm $\mathcal{Q}$ with virtual processing using $p \cdot vpr$ VPE's and let $s_p(\mathcal{Q})$ be the time to run $\mathcal{Q}$ without virtual processing using p physical PE's. Then

$$t_p(\mathcal{Q}, vpr) = s_p(\mathcal{Q}) + vpr \cdot l_p(\mathcal{Q}) + c(\mathcal{Q}),$$

where $c(\mathcal{Q})$ is the time to initialize and to wrap-up the simulation of virtual processors for algorithm $\mathcal{Q}$ and $l_p(\mathcal{Q})$ is the unit time used in performing local computations in the simulation. Since we expect $s_p(\mathcal{Q}) > l_p(\mathcal{Q})$ for the MasPar MP-1, routines in this category had super-linear speed-ups when using virtual

processors (i.e., $\frac{t_p(\mathcal{Q},vpr)}{s_p(\mathcal{Q})} < vpr$). In Table 12.1, we also show the ratio between the performance of our current implementation with virtual processing and the performance of a previous implementation in Chapter 11 without virtual processing using $p = 16{,}384$ physical processors. Note that this ratio is roughly equal to $vpr \cdot \frac{l_p(\mathcal{Q})}{s_p(\mathcal{Q})}$.

On a massively parallel computer, we would expect the value of $s_p(\mathcal{Q})$ to be smaller as p becomes smaller. Thus the best way to compute algorithm $\mathcal{Q}$ with virtual processing on an input of size k is not to use all of the available physical processors. Instead, one should use only r physical processors such that $t_r(\mathcal{Q}, d)$ plus the time to evenly distribute the data is minimized, where $k = r \cdot d$. However, we found that $s_p(\mathcal{Q})$ remained the same no matter how many active physical processors were used in the MasPar MP-1 for all the routines in this category. (This fact was also confirmed by experiments conducted in [Pre93b].) Thus we were unable to further fine-tune our code using this approach.

12.4.2 Category 2

Parallel operators in this category could be implemented with virtual processing by exactly vpr executions of the original parallel primitives (without virtual processing) and some local computations performed in each physical processor. The following parallel primitives are in this category.

- The *routing* operator. The `router` operators are used for performing global routing. For example, the MPL system routine `router`[*addr*].*data* returns the value of the plural variable *data* that is stored in the processor with ID *addr* for each active processor.

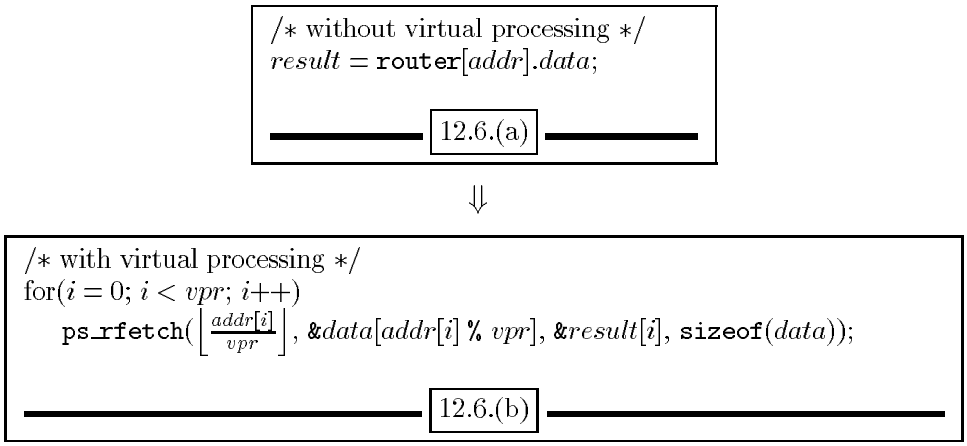
- The *list ranking* operator. Given a linear linked list, each element in the list is stored in a processor. For each element, compute the summation

of all the data that are ahead of it in the linked list. This operator is described in the kernel routines given in Chapter 11. For example, the kernel routine `listrank32`(*ptr*, *data*) performs list ranking on the plural variable *data* for each active processor, where *ptr* specifies the location of the next element.

We now describe our implementations of primitives in category 2 and their performance data in the following subsections.

Routing

Implementation The MPL language provides an efficient implementation of the routing operator if the data needed to be routed is any one of the fundamental data types allowed by the MPL language (e.g., not a member of an array element). However, to implement routing operators using virtual processing, we needed to access different indices of a plural array within each physical processor. To do this, we used the MPL system routine `rfetch`(*from*, *st*, *to*, *sz*) to retrieve *sz* bytes of data starting from the local address *st* of the physical processor with ID *from* and store it in the local address *to* for each active processor. For example in Program 12.6.(a), each processor fetched the data stored in the processor with ID *addr* and stored it in the local variable *result*. The size of the data was `sizeof`(*data*) bytes. (Note that `sizeof` is an MPL system routine.) To implement this statement with virtual processing, we had to access the proper data element allocated to each virtual processor in our implementation. We show the implementation of a routing statement with virtual processing in Program 12.6.(b). After testing our program, we found that the above implementation did not work properly when $vpr = 1$ because



Program 12.6: MPL code segments for routing statements. Note that "%" is the modulo operator in the MPL language and "&" is an unary operator to compute the address of a variable. Note also that the data allocated to the VPE with ID $addr[i]$ was stored in the $(addr[i]$ % $vpr)$th element of the plural array $data$ in the physical processor with ID $\lfloor \frac{addr[i]}{vpr} \rfloor$.

the MPL system routine `rfetch` does not function correctly when its input parameters are arrays of size one. Thus most of our programs reported in this chapter that used the router did not work for $vpr = 1$.

Further Fine-Tuning The performance of a routing statement in the MasPar MP-1 is determined by the following three factors: the size of data transmitted by each active processor, the number of virtual processors that are active, and the maximum degree of concurrency among all physical processors. The first two factors are related to the bandwidth of the communication network and the third factor is related to the sequentialization of concurrent requests on physical processors. The first two factors are determined by the nature of the routing request and are hard to be changed.

To improve the performance of a routing statement, we could reduce the maximum degree of concurrency (i.e., the third factor). There is a well-known PRAM algorithm to simulate a concurrent read (write) statement by

using sorting and exclusive read (write) statements [Eck79, Vis83, KR90]. Although this algorithm does not eliminate all concurrency in reading (writing) when virtual processors are used (i.e., reading (writing) to VPE's with different ID's might be mapped into different locations in the same physical processor), one would expect the maximum degree of concurrency to be greatly reduced. Note that we need to pay the overhead of performing a sorting routine to use this algorithm.

We implemented the above algorithm for routing and tested our program when $vpr = 16$ while varying the expected degree of concurrency. We also tested Program 12.6 with virtual processing and $vpr = 16$ and the system routing routine without virtual processing. Let con be the expected degree of concurrency in our testing (i.e., the expected number of physical (virtual) processors that read data from a physical (virtual) processor without (with) virtual processing). We obtained the desired expected degree of concurrency by drawing the destination addresses from a system pseudo-random number generator in the range from 0 to $\frac{nproc}{con}-1$ without virtual processing, and in the range from 0 to $\frac{vpr \cdot nproc}{con} - 1$ with virtual processing, where $1 \leq con \leq nproc$. Note that it is well-known (for example, it is fairly easy to derive from [ASE92, Che52]) that with high probability the maximum number of requests on any physical processor is some constant times con when $con \geq \log(\frac{nproc}{con})$ without virtual processing and is some constant times $vpr \cdot con$ when $vpr \cdot con \geq \log(\frac{nproc}{con})$ with virtual processing. Thus con is the expected degree of concurrency per virtual processor if it is sufficiently large.

The performance data is shown in Table 12.2. Table 12.2 suggested that the performance of the system routine for performing concurrent read on a physical processor did not grow linearly in the number of expected concurrent

expected concurrency (con)	Routing 32-bit Integers time (ms)		
	no virtual processing	$vpr = 16$	
		Program 12.6	convert to exclusive read
1	0.45	18.00	236.97
2	0.67	26.95	239.00
4	1.13	45.03	244.78
8	1.56	63.56	238.61
16	2.86	115.71	237.72
32	5.33	218.57	236.51
64	5.44	216.33	232.18
128	5.49	218.23	230.30
256	10.59	415.99	233.74
512	20.60	819.08	229.88
1,024	41.28	1,624.66	229.56
2,048	82.94	3,250.92	229.30
4,096	144.84	5,673.66	231.47
8,192	144.84	5,675.19	231.47
16,384	144.84	5,676.73	234.77

Table 12.2: Comparison of performance data for reading 32-bit integers from different processors without virtual processing and with virtual processing when $vpr = 16$ on the MasPar MP-1. The number of physical processors used is $nproc = 16{,}384$. We tested two implementations of inter-processor memory accessing with virtual processing. The destination addresses were drawn from a system pseudo-random number generator in the range from 0 to $\frac{nproc}{con} - 1$ without virtual processing, and in the range from 0 to $\frac{vpr \cdot nproc}{con} - 1$ with virtual processing.

requests. Instead, the running time fell into the repeated patterns of growing linearly for some values of con and then staying unchanged for other values of con. We also notice that the naive implementation with virtual processing shown in Program 12.6 ran faster than its exclusive read version when con was within 128. However, for programs that expect a high degree of concurrency, the exclusive read version should be used.

Permutation Routing We also tested the performance of a permutation routing with and without virtual processing. The performance data is shown in Table 12.3. Note that the expected degree of concurrency on a physical processor is proportional to the value of vpr when virtual processors are used.

	Permutation Routing	
	time (ms)	ratio
system (no VPE)	0.45	
$vpr =$ 2	2.28	5.07
4	4.57	10.16
8	9.25	20.56
16	18.12	40.27
32	36.53	81.18
64	72.33	160.73

Table 12.3: Performance data for performing a permutation routing of 32-bit integers. The "ratio" column gives the ratio of the time used by the routine using *vpr* virtual processors per physical processor to the time used by the non-virtual processing routine.

Thus when the value of *vpr* doubled, we not only had twice the number of virtual processors, but also had twice the expected degree of concurrency on each physical processor. The performance could be 4 times as bad when we doubled the value of *vpr*. However in Table 12.3, we notice that that the cost for performing a permutation routing was linear in the value of *vpr* instead of being quadratic. From the data obtained in this section, we can conclude that if the degree of concurrency is not too large, it is better to use the naive implementation of routing given in Program 12.6.

List Ranking We implemented the simple EREW list ranking algorithm using pointer jumping [KR90]. The algorithms runs in $O(\log n)$ iterations by repeatedly changing the current pointer of each active processor to the pointer stored in the processor that is currently next to it. A processor becomes inactive when it reaches the end of the list. The MPL code for this algorithm is shown in Program 12.7. Note that each iteration of the **for** loop in steps 1 and 2 of Program 12.7.(b) required performing a router operation. Note also that steps 1 through 3 in Program 12.7.(b) corresponded to steps 1 and 2 in Program 12.7.(a). To transform steps 1 and 2 in Program 12.7.(a) to handle virtual processing, temporary variables were needed to store the intermediate

```
/* without virtual processing */
/* Perform a list ranking on the linked list specified by the
the successor pointer ptr and the element data in each processor.
The pointer value of the last element is negative.
The result will be stored in rank. */
int listrank32(rank, ptr, data)
plural int rank, ptr, data;
{
    plural int pointer;;
    rank = data; pointer = ptr;
    while there is a value in pointer that is not negative{
1.      rank = rank + router[pointer].rank;
2.      pointer = router[pointer].pointer;
    };
}
```

12.7.(a)

⇓

```
/* with virtual processing */
int vlistrank32(rank, ptr, data, active)
plural int rank[vpr], ptr[vpr], data[vpr]; plural char active[vpr];
{
    register int i; plural int tmpptr[vpr], temp, pointer[vpr];
    /* the first element in the list has a negative pointer value */
    for(i = 0; i < vpr; i++)
        if(active[i]){
            pointer[i] = ptr[i]; rank[i] = data[i];
        }else pointer[i] = −1;
    while there is a value in pointer that is not negative{
        for(i = 0; i < vpr; i++)
1.          if(active[i] and pointer[i] > 0){
                ps_rfetch(⌊pointer[i]/vpr⌋, &rank[pointer[i] % vpr], &temp, sizeof(rank));
                rank[i] += temp;
            }
        for(i = 0; i < vpr; i++)
2.          if(active[i] and pointer[i] > 0)
                ps_rfetch(⌊pointer[i]/vpr⌋, &pointer[pointer[i] % vpr], &tmpptr[i],
                sizeof(pointer));
3.      for(i = 0; i < vpr; i++) if(active[i]) pointer[i] = tmpptr[i];
    }
}
```

12.7.(b)

Program 12.7: The MPL code for a simple sub-optimal deterministic list ranking algorithm with virtual processing.

	listrank32		
	not pipelined	pipelined	
	time (ms)	time (ms)	ratio
use no VPE		13.91	
vpr = 2	72.05	44.16	3.2
4	154.55	94.41	6.8
8	330.37	201.43	14.5
16	704.30	429.10	30.8
32	1,494.33	910.17	65.4
64	3,164.83	1,927.21	138.5

Table 12.4: Performance data for performing the list ranking operation on 32-bit integers. The "ratio" column gives the ratio of the time used by the routine using *vpr* virtual processors per physical processor to the time used by the non-virtual processing routine.

results. Thus we expected the amount of memory used by our programs with virtual processing to be more than *vpr* times the amount of memory in the programs for non-virtual processing routines.

Pipelined Requests Note that in steps 1 and 2 of Program 12.7.(b), each VPE retrieved *data* and *pointer* from the same VPE. We observe that it is often the case on the MP-1 that the total time required for two inter-processor communications of l_1 and l_2 bytes, respectively, is more than the time required for one inter-processor communication of $l_1 + l_2$ bytes. Since each VPE read from the same VPE in steps 1 and 2, we could pipeline these two requests. By doing this, we were able to save time, though programming pipelined operations required additional temporary space to pack data together. We list the performance of our list ranking algorithms (both pipelined and non-pipelined) in Table 12.4.

Note that in Table 12.4, we did not list the performance of the pipelined list ranking without virtual processing because the overhead was larger than the benefit. We also did not obtain performance data for the case when $vpr = 1$, since our current implementation did not work for this case.

(The reason is explained earlier in the previous section.) We notice that the usage of pipelined instructions made our program run about 1.6 times faster than the non-pipelined version in the case when $vpr = 64$. Since the amount of temporary space used in implementing pipelined requests was relatively small, we decided to use the pipelined version for our code.

Randomization We did not implement the list ranking operator using the optimal deterministic $O(\log n)$-time $O(\frac{n}{\log n})$-processor algorithm presented in [CV88, AM88], where n is the number of elements in the list, since we felt that the coding would be tedious and the overhead would be too large. Instead, we implemented the simple $O(\log n)$-time $O(n)$-processor deterministic algorithm which is shown in Program 12.7.(b). Let *nporc* be the number of physical processors. In the case of $nproc \geq n$, Program 12.7.(b) is optimal and should run faster than the algorithms described in [CV88, AM88] because of simplicity. We expect Program 12.7.(b) to run slower when n is greater than *nproc*.

A Simple Optimal Randomized Algorithm In addition to the fairly complicated deterministic optimal algorithms in [CV88, AM88], there are several simple randomized optimal algorithms for performing the list ranking operator (see e.g., [KR90, Vis84, CLR90, AM86]). A simple randomized optimal list ranking algorithm [CLR90] is shown in Algorithm 12.1. The implementation of step 1.1 on the MasPar using the MPL needed to use one global routing of 8-bit flags. Step 1.2 used one global routing of 8-bit flags and two pipelined global routings of 32-bit integers. Step 2 used only one global routing of 32-bit integers. All global routings used in this algorithm were done among physical processors.

```
/* An optimal randomized list ranking algorithm. */
1.  /* Shrink the size of the list */
    mark all elements "un-eliminated"
    assume vpr = n/nproc is an integer
    assigned vpr elements to a PE
    while the total number of "un-eliminated" elements > 1 do
       for each PE_i execute in parallel
          pick an "un-eliminated" element e_i
          generate a random bit b(e_i) for e_i
          let s(e_i) be the successor of e_i in the current list
1.1.      if b(e_i) = 1 and (b(s(e_i)) = 0 or no random bit assigned for s(e_i))
1.2.      then eliminate e_i from the list
       end
    end
2.  compute the rank of the eliminated element by "unraveling" the computation
    of step 1
```

Algorithm 12.1: An optimal randomized list ranking algorithm.

Recall that each iteration in the simple list ranking algorithm (Program 12.7.(b)) used two pipelined global routings of 32-bit integers among virtual processors. (Routing among virtual processors was implemented using Program 12.6.) The performance data for Program 12.7.(b) and our MPL code for Algorithm 12.1 are shown in Table 12.5. In our testing, we terminated our program if more than 512 iterations were needed to finish step 1 of Algorithm 12.1. We marked those experiments that needed more than 512 iterations as failure because both the time spent and the extra space used were too large. Although the average time needed for each successful trial was smaller than the time needed for Program 12.7.(b) when $vpr \geq 16$, the probability of success for Algorithm 12.1 was fairly large. The reason might be that the size of the input is not large enough and hence the constant factor in the theoretical analysis [CLR90] is large.

By tracing our code, we observed the following. If the total number of elements was significantly smaller than the number of physical processors, then only a fairly small percentage of elements were eliminated during each iteration.

However, if the total number of elements was larger than the number of physical processors, then almost *nproc* elements were eliminated during each iteration. (Note that we could eliminate at most *nproc* elements during each iteration.) We conjecture that the reason for this behavior might come from the MPL system pseudo-random number generator `p_random` used in our code. The system routine `p_random` generates a pseudo-random number for each active PE. On Page 5 of [Mas92d], it says "Although the values of `p_random` are random sequences from each PE's point of view, you might see some discernible pattern in the PE array as a whole." Since the behavior of the pseudo-random number degrades when one only use a small portion of the numbers generated by all the PE's, our performance data did not match the bound obtained by the theoretical analysis. It would be interesting to obtain a good pseudo-random number generator for SIMD parallel computers to produce good pseudo-random numbers among all processors.

A Revised Optimal Randomized Algorithm In order to remedy the above problem possibly caused by the usage of the system pseudo-random number generator, we observe that our simple deterministic algorithm (Program 12.7.(b)) is optimal if $nproc \geq n$. Thus we modified Algorithm 12.1 as follows. We ran the randomized algorithm until the first time the number of "un-eliminated" elements was no more than the number of physical processors. Then we ran the deterministic algorithm without virtual processing given in Chapter 11 by evenly distributing the remaining elements among available physical processors. Each physical processor would get at most one of the remaining elements. We finally "unraveled" the computation and obtained the result for each virtual processor. The algorithm is shown in Algorithm 12.2.

Since the number of remaining elements in each iteration was at least equal to the number of physical processors during the execution of step 1 in Algorithm 12.2, the probability that one physical processor retained at least one "un-eliminated" element was very high. Thus we could eliminate a fairly large amount of elements in each iteration even using the system pseudo-random number generator. The performance data for Algorithm 12.2 is also shown in Table 12.5. Note that the number of iterations needed to execute Algorithm 12.2 is much smaller than the number of iterations needed to execute Algorithm 12.1. All of our test cases for Algorithm 12.2 succeeded in $\frac{5}{2} \cdot vpr$ iterations. We also note that on the largest input, Algorithm 12.2 was more than twice faster than the sub-optimal deterministic version. The drawback of Algorithm 12.2 is that the implementation of step 2 was non-trivial and it used more than thrice the amount of space than Program 12.7.(b) for storing the history of step 1 in order to "unravel" the computation in step 3. When memory is at a premium, Program 12.7.(b) should be used. (The implementation of a set of parallel graph algorithms as described in Chapter 12 used Program 12.7.(b) to save space.) However, for machines that provide large amount of local memory for each virtual processor (e.g., the Connection Machine), Algorithm 12.2 should be used.

12.4.3 Category 3

Parallel primitives in this category could not be implemented with virtual processing by the simple strategies given in Section 12.4.1 or Section 12.4.2. For example, giving a parallel operator that sorts $nproc$ elements does not imply that one can apply this operator vpr times to sort $vpr \cdot nproc$ elements. We have to look for different algorithms for solving these problems instead of rely-

/* A modified optimal randomized list ranking algorithm. */
1. /* Shrink the size of the list */
   perform the **while** loop in step 1 of Algorithm 12.1 until the the total number of remaing elements is $\leq$ the number of physical processors
/* Load Balancing */
2. /* compute the rank of the current list by using the simple algorithm specified in Program 12.7.(b) */
2.1 distribute the remaining "un-eliminated" elements evenly among physical processors
2.2 compute the rank using Program 12.7.(b)
2.3 send results back to original processors
3. compute the rank of the eliminated element by "unraveling" the computation of step 1

Algorithm 12.2: A modified optimal randomized list ranking algorithm.

vpr	Deterministic Algorithm		Randomized Algorithms					
	Program 12.7.(b)		Algorithm 12.1			Algorithm 12.2		
	time (ms)	# iterations	% success	time (ms)	# iterations	% success	time (ms)	# iterations
2	44.16	15	42.5	179.96	150.7	100.0	31.63	3.6
4	94.41	16	45.0	260.00	218.2	100.0	54.06	9.9
8	201.43	17	47.5	266.58	174.8	100.0	92.14	15.7
16	429.10	18	70.0	409.08	207.6	100.0	180.19	37.1
32	910.17	19	62.5	642.37	237.5	100.0	371.36	74.0
64	1,927.21	20	40.0	1,134.99	287.7	100.0	808.53	145.5

Table 12.5: Performance data for the three algorithms we implemented for performing list ranking deterministically with virtual processing. For each algorithm, we also record the average number of iterations performed in Program 12.7.(b) and in step 1 of Algorithm 12.1 and Algorithm 12.2 over 40 different trials. Randomized algorithms were terminated and marked as failure if more than 512 iterations were needed.

ing on iteratively applying the basic parallel primitives. The following parallel primitives are in this category.

- The *sort* operator which sorts elements in parallel. For example, the MPL system routine `psort`(*result*, *data*) sorts *data* in each processor and stores them in *result*.

- The *rank* operator which computes, for each processor, the number of elements among all local copies of a plural variable d that are smaller than the processor's local value of d (break ties arbitrary). The value

computed is its rank.

For example, the MPL system routine `prank`($result, data$) stores the rank of $data$ in $result$ for each processor.

- The *sendwith* operator which combines data sent to each processor. To use the *sendwith* operator, one specifies an associative operator, a source plural variable s, and a plural variable d for destination address. Each active processor sends the local value of s to the processor whose ID is specified in the local value of d. If there are several values sent to one processor, all incoming values (this does not include the original value in the destination processor) are combined using the given associative operator before storing into the destination processor.

  For example, the MPL system routine `sendwithAdd32`($result, data, addr$) sends the local value of $data$ in each processor to the local variable $result$ in the destination processor $addr$. If several data are sent to one processor, then the summation of these data is stored.

- The *range minimum* (*maximum*) operator. Given local variables v, s and t in each processor, we build data structures such that we can return the minimum (maximum) value of all v's from the processor with ID s to the processor with ID t for each processor. This operator is described in the kernel routines given in Chapter 11. For example, the kernel routine `buildTMin32`(v) builds such data structures for 32-bit integers and the kernel routine `queryMin32`(s, t) returns the minimum (maximum) values queried. We implemented these two routines in the work reported in Chapter 11 without virtual processing.

We describe these primitives in the following.

Sort, Rank, and Sendwith The first three routines in this category could be implemented with virtual processing by calling a sorting routine that could handle virtual processors. For sorting with virtual processing, we used the package developed in [PS90]. Since the sorting package in [PS90] can only handle the case when the number of virtual processors simulated by each physical processor is a power of two, our code inherited the same restriction.

To implement the `rank` operator with virtual processing on the plural array a, we first created a 2-tuple (a, $viproc$) for each VPE, where $viproc$ was its ID. We then sorted these 2-tuples lexicographically. After sorting, each VPE sent its current rank value back to the VPE that originally had the a value before the sorting (whose ID was indicated in the second component of the 2-tuple). Note that to compute the ranks of 32-bit integers, we had to sort 64-bit data.

The implementation of the `sendwith` operator with virtual processing is as follows. We first sorted all requests from active VPE's according to their destination addresses. We then grouped requests to the same destination in a segment and performed a segmented scan operation [Ble89] using the associative operator specified by the `sendwith` operator. In each segment, we picked the last VPE to write the result to the destination. Note that to send 32-bit integers using the `sendwith` operator, we had to sort 64-bit data.

Performance data for the first three routines in this category is shown in Table 12.6. Note that the sorting routine that we used is very competitive compared to the system sorting routine. For sorting $vpr \cdot nproc$ elements, the time required is about vpr times the time used by the system sorting routine which sorts $nproc$ elements. Note also that the overhead for performing the `sendwith` routine with virtual processing was pretty large. To send $vpr \cdot$

	psort32		prank32		sendwithMax64	
	time (ms)	ratio	time (ms)	ratio	time (ms)	ratio
system (no VPE)	7.69		9.44		9.6	
$vpr =$ 2	17.87	2.3	24.76	2.6	34.4	3.6
4	31.07	4.0	48.75	5.2	66.9	7.0
8	59.64	7.8	98.57	10.4	134.4	14.0
16	118.99	15.5	201.68	21.4	273.8	28.5
32	241.90	31.5	415.41	44.0	559.8	58.3
64	496.62	64.6	860.70	91.2	1,149.9	119.8

Table 12.6: Performance data for performing the `sort` operator [PS90] and the `rank` operator on 32-bit integers, and for performing the `sendwith` operator on 64-bit integers by using maximum as the combining operator. The "ratio" columns give the ratio of the time used by the routines using vpr virtual processors per physical processor to the time used by the non-virtual processing routines.

$nproc$ elements with virtual processing, we had to spend about $2 \cdot vpr$ times the amount of time to send $nproc$ elements without virtual processing. The overhead for implementing the `rank` operator with virtual processing was in between the overhead for implementing the `sort` operator and the overhead for implementing the `sendwith` operator.

Range Minimum (Maximum) Recall from Chapter 11, the table we built in each processor for the range minimum (maximum) queries without virtual processing was a one-dimensional array W such that $W[i]$ is the minimum (maximum) value of all elements starting from the current processor to a processor whose ID was $2^i - 1$ larger than the ID of the current processor. To retrieve the minimum (maximum) value starting from the processor with ID s to the processor with ID t, we returned the minimum (maximum) value of $W[k]$ in the processor with ID s and $W[k]$ in the processor with ID $t - 2^k + 1$, where $2^k \leq t - s + 1 < 2^{k+1}$. We could implement this with virtual processing using a straight forward approach of building such an array for each virtual processor. However, the size of the array in each physical processor would be $vpr \cdot \lceil \log(vpr \cdot nproc) \rceil$.

Instead of the above strategy, we used the following approach. For each physical processor that simulated virtual processors with ID's from x to $x + vpr - 1$, we built a suffix array ST and a prefix array PT such that $ST[i]$ was the minimum (maximum) value of elements from $x + i$ to $x + vpr - 1$ and $PT[i]$ was the minimum (maximum) value of elements from x to $x + i - 1$. We also built a two-dimensional inner array IT such that $IT[i, j]$ was the minimum (maximum) value of all elements in virtual processors with ID's from $x + i$ to $x + i + 2^j - 1$, for all j such that $2^j < vpr$. Let v be the minimum (maximum) value of all elements in virtual processors simulated by one physical processor. We built a global array W for v's that was the same as the array we built without virtual processing.

To query the minimum (maximum) value of a range of elements allocated on virtual processors that were simulated by one physical processor, we used the inner array IT. To query the value of a range of elements allocated on virtual processors simulated by more than one physical processor, we used the suffix array ST, the prefix array PT and the global array W. The total size of arrays in each physical processor was $\lceil \log(nproc) \rceil + vpr \cdot (2 + \lceil \log(vpr) \rceil)$. Although our implementation issued four routing requests per virtual processor, two more than the naive approach, to perform the queries, our approach used less memory than the naive approach when $nproc \geq 4$. For large values of $nproc$, the above approach used about $\frac{1}{vpr}$ of the amount of memory used by the naive approach.

Thus for parallel programs that require a lot of memory, the above approach should be used. For massively parallel computers with a larger memory space per virtual processor, the naive approach should be used to achieve up to two times speed-up. We show the performance data for `buildMinT32`

	buildMinT32	
	time (ms)	ratio
system (no VPE)	0.036	
vpr = 2	0.039	1.08
4	0.041	1.14
8	0.048	1.33
16	0.070	1.94
32	0.135	3.75
64	0.361	10.02

Table 12.7: Performance data for building data structures to support range minimum queries of 32-bit integers. The "ratio" column gives the ratio of the time used by the routine using *vpr* virtual processors per physical processor to the time used by the non-virtual processing routine.

in Table 12.7. Note that the performance of our range minimum query routine depends on too many parameters (e.g., the degree of concurrency on the queries). Thus we did not obtain its performance data.

12.5 Comparisons Between Sequential and Parallel Implementations of Basic Primitives

In this section, we compare the performance data for sequential and parallel implementations of three basic parallel primitives, one each from the three categories described in Section 12.4. The three basic parallel primitives are prefix summing, list ranking and sorting.

We implemented the above three operators sequentially using the C programming language [KR88] on a UNIX system [RT74] (SunOS 4.1.3). We used the quick sort routine provided by the system library for sorting. The performance data for the sequential algorithms was obtained by running our programs on a SPARC 10/41 machine with 32 megabytes of main memory and about 80 megabytes of swapping space. As indicated in Chapter 11, the SPARC II is at least 200 times faster than a single MasPar PE. We ran our sequential code for finding connected components on our SPARC 10/41 and compared its running time with the running time used on a SPARC II. We

found that SPARC 10/41 is at least 1.15 times faster than a SPARC II. Thus SPARC 10/41 is at least 230 times faster than a single MasPar MP-1 PE. Since the MasPar MP-1 that we were using had 16,384 PE's, the raw computational power of the MP-1 was at least 63 times larger than a SPARC 10/41.

All of our performance data for sequential programs was obtained when the machine had only one active job. We measured two quantities in performing sequential computations. The first quantity was the user time, which is the amount of the CPU time used directly by a computation. The second quantity was the total time, which is the user time plus the amount of time used by the system to handle events generated while executing a computation, e.g., the swapping between the cache system and the main memory when there is a cache miss, and the swapping between main memory and the secondary storage when there is a page fault. On a single user system, the total time of a sequential computation is roughly equal to its turn-around time, i.e., the wall clock time during the computation.

We found that for running our sequential programs on inputs that required less than 24 megabytes (or 80% of the main memory on our system), the total time was roughly equal to the user time. When the amount of memory used was more than 32 megabytes, the effect of slow down due to swapping started to appear. For inputs that required $24 - 32$ megabytes of memory, the behaviors of our programs varied. One advantage in using a massively parallel computer, in addition to the enormous raw computation power offered, is the huge total amount of main memory available for a computation. For example, although each physical processor in the MasPar MP-1 only had 64 kilobytes of memory, the total available memory for a 16,384-processor machine was 1 gigabytes (or 32 times larger than the main memory of our SPARC 10/41).

Thus we would expect that our parallel implementations can run larger inputs faster than our sequential implementations, since sequential implementations would spend most of the time swapping when the input could not fit into the main memory. We expect the total available main memory would become even larger on a parallel machine once the speed of a sequential machine reaches its limit. At that time, our parallel implementation would be able to run much larger inputs.

The performance data for performing the prefix summing, list ranking and sorting are shown in Figure 12.2, Figure 12.3, and Figure 12.4, respectively. On the left of each figure is the performance data for the sequential implementation. On the right of each figure is the performance data for the parallel implementation and a curve obtained by using a least-squares-fit package in Mathematica [Wol88]. We also obtained a least-squares-fit curve for the sequential performance data when the amount of memory used in the program was within the limit of the main memory.

12.5.1 Prefix Sums

Programs in the first category issued regular communication requests to nearby processors in their parallel implementations. These programs generated a linear number of swapping requests on the part of the data that was placed on the secondary storage when implemented sequentially if the locality of data referencing in the programs was utilized wisely. For example, each data would have to be placed in the main memory (or cache) only once to compute prefix summing sequentially. Thus we would expect to achieve a linear speed-ups on programs in this category in their parallel implementations. We observe that our parallel implementation ran about 70 times faster than our

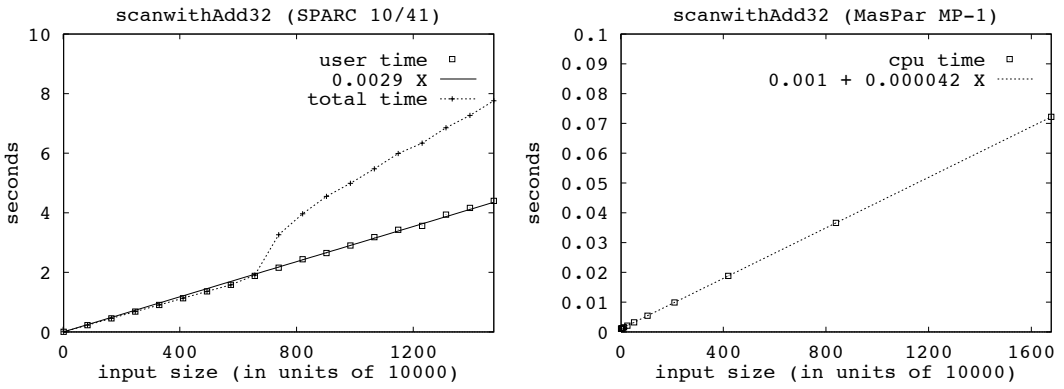


Figure 12.2: Performance data for performing prefix summing sequentially and in parallel.

sequential implementation when all data could fit in the main memory of our SPARC 10/41. Note that we only expected the MasPar MP-1 to be 63 times faster than the SPARC 10/41. When data had to be placed out of the core in our sequential code, our parallel program ran about 100 times faster than our sequential implementation on the largest input that we have tested. The total amount of swapping time was about 3 seconds, which was about 40% of the total time, on the largest input that we have tested.

12.5.2 List Ranking

Programs in the second category had unpredictable communication patterns in their parallel implementations. The amount of speed-ups that one could get was thus proportional to the performance of the inter-processor communication primitives provided by the parallel machine. On the other hand, the sequential implementation of these programs generated a large number of swapping requests when part of its data had to be placed out of core since very little locality could be used for data accessing. Thus we would expect programs in this category to have very little speed-ups when the size of the inputs was

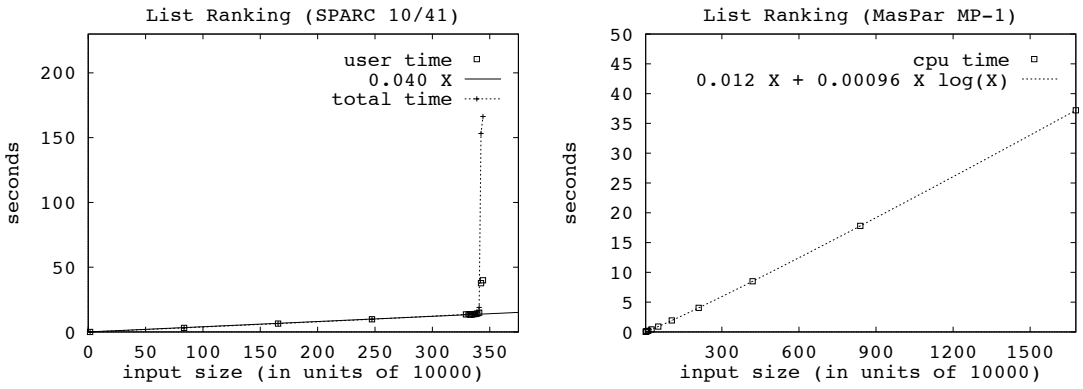


Figure 12.3: Performance data for performing list ranking sequentially and in parallel.

small and to have super-linear speed-ups when the size of the inputs was large.

It is well-known that that data in the secondary storage of a sequential machine (e.g., SPARC 10/41) is partitioned into fixed size segments, e.g., 512-byte segments or 1,024-byte segments. When a program references a cell that is not currently in the main memory, the segment where the cell is in is brought in by swapping an equal size segment out. The same scheme is performed in the cache level. In the case of performing list ranking on large inputs, the number of times a segment had to be brought in from the secondary storage is proportional to the size of the segment. For example, when half of the elements were in the secondary storage, our program had to generate a page fault about every two memory references. From the performance data shown in Figure 12.3, we observe that our parallel implementation was only twice faster than our sequential implementation if all cells in the linked list were in the main memory. However, when heavy swapping was needed in the sequential implementation, its performance degraded dramatically. We tested our code on larger and larger inputs until the wall-clock time for finishing a job was more than 24 hours on our SPARC 10/41 when there was only one active job in the system. For the

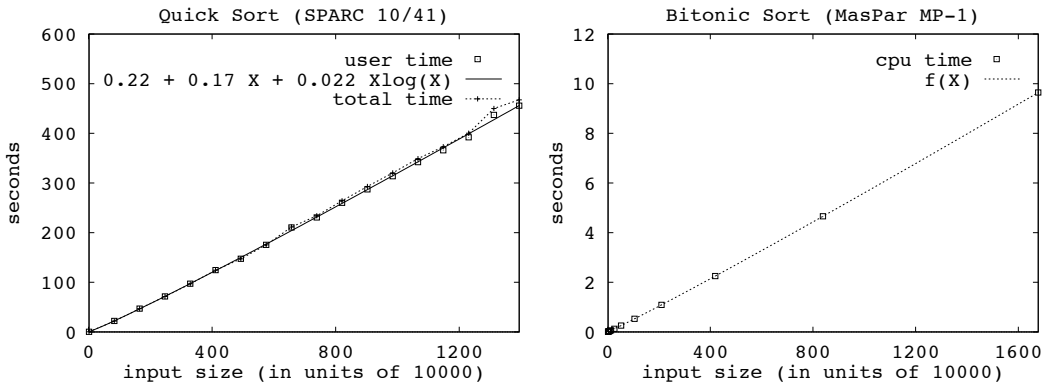


Figure 12.4: Performance data for performing sorting sequentially and in parallel. The sequential implementation used a quick sort algorithm. The parallel implementation which is given in [PS90] uses a bitonic sort algorithm. The function $f(X)$ in the right figure is $0.004 + 0.004X + 0.001X \log X + 0.0000064X \log^2 X$.

largest input that we tested, the sequential program spent over 120 seconds, which was more than 75% of the total time, in performing swapping. On the other hand, our parallel implementation performed predictably well for the size of the the input that was more than 5 times larger than the largest input that we have tested on the sequential implementation.

12.5.3 Sorting

Programs in the third category issued predictable communication requests to processors that were not near-by in their parallel implementations. Their sequential implementations generated a moderate number of swapping requests when part of their input had to be placed out of core. Thus we would expect a moderate speed-up in their parallel implementations. For example, we would expect each segment in the secondary storage to be swapped only a logarithmic number of times in a good sequential implementation of the quick sort algorithm. Our performance data in Figure 12.4 shows that the swapping time

for the largest input on our sequential implementation was about 12 seconds, which was only about 2.5% of the total time. The parallel implementation of the bitonic sort algorithm we used [PS90] was about 45 times faster than the sequential implementation of the quick sort algorithm on the largest input that we have tested.

12.6 Concluding Remarks

We have described our techniques for implementing virtual processing on the MasPar MP-1 using the MPL language. We have described our data allocation and code rewriting rules for writing MPL programs such that the number of processors used in the program is not limited. We have also described the implementation and fine-tuning of a set of parallel primitives with virtual processing. We will build on this approach in Chapter 13 where we present the implementation of a set of efficient parallel graph algorithms.

We note the following observations.

- The current architecture of the MasPar MP-1 is not adequate to run programs that require a lot of memory per physical processor. The router is too slow (about 200 times slower) compared to the speed of a mesh communication. It is reported in [Pre93a] that the new MasPar MP-2 upgrades the raw computation power of each individual processor while keeping its communication hardware and limitation of memory space unchanged. For our application, we feel that the amount of memory in each processor should be increased and the bandwidth of the communication channel should be enlarged before the upgrading of the processor computation power.

- Our approach of implementing virtual processing on the MasPar MP-1 using the MPL language is promising. The performance of parallel primitives after fine-tuning was predictable for the range of vpr, the number of virtual processors simulated by each physical processor, that we have tested. We also observe that for the set of parallel primitives that we have implemented, there is a strong relationship between the swapping time used by our sequential code and the inter-processor communication time used by our parallel code.

- Amdahl's law for parallel processing [Amd67] states that if $\frac{1}{x}$ of the computation in a program can only be speeded up by a factor of y, then the maximum speed-ups that one can get is $x \cdot y$. For designing parallel programs with virtual processing by calling parallel primitives implemented in this chapter, one obtains the least speed-ups (less than 2 times) from primitives in category 2 (Section 12.4.2) when the size of input can fit into the main memory of the sequential machine. Thus we would expect only 2 to 3 times speed-ups when the size of input is small in parallel programs that require to spend a constant fraction of their computation time on routines in category 2. However, for large inputs, we can achieve large speed-ups. This has been confirmed by the experiments conducted in the next chapter for implementing a set of parallel graph algorithms with virtual processing using the approach outlined in this chapter.

Chapter 13

Implementation of Parallel Graph Algorithms on a Massively Parallel SIMD Computer with Virtual Processing

13.1 Introduction

This chapter continues the discussion of our implementation project on the massively parallel computer MasPar MP-1. There has been a fair amount of prior work for implementing parallel algorithms on massively parallel machines [AS92, BLM+91, DL92a, DL92b, HPR92, NT92, PS90] since the completion of the first phase of our project reported in Chapter 11. However, most of this work has been targeted towards solving problems that are highly structured and are not very difficult to scale up. The focus of our work is on solving graph-theoretical problems for which the algorithms require large amounts of non-oblivious memory accesses.

In Chapter 11, we described the implementation of several parallel graph algorithms on the MasPar MP-1 using the parallel language MPL [Mas92b, Mas92c] which is an extension of the C language. The MPL language provides a very efficient way of using the MasPar with the drawback of requiring the specification of the physical organization of the processors used in the program. Our implementation described in Chapter 11 used an edge list data structure to store the input graph. An undirected edge (u, v) was stored twice as one directed edge from u to v and another directed edge from v to u. Each of the two copies of an undirected edge was stored in one processor along

with a node. As a result, we could only handle the case when the input graph has no more than $nproc$ nodes and $\frac{nproc}{2}$ edges where $nproc$ is the maximum number of processors that we can use in the system. The machine that we used, the MasPar MP-1, had $nproc = 16{,}384$ processors. In this chapter, we report the second phase of this work, which consisted of implementing these graph algorithms to handle inputs of size greater than 16,384 using the techniques developed in Chapter 12. In this current implementation, we use the rewritting rules and the library routines given in Chapter 12 to obtain our code with virtual processing.

Our results are reported in the following sections which are organized as follows. Section 13.2 gives a high-level description of our implementation. Section 13.3 describes the implementation details of our parallel graph algorithms library. Section 13.4 gives performance analysis. Finally, Section 13.5 gives the conclusion. The work presented in this chapter also appears in [HRD93].

13.2 High-Level Description of Our Implementation

In our earlier implementation of parallel graph algorithms without virtual processing described in Chapter 11, we first provided a general mapping between the architecture of the MasPar and the schematic structure of the PRAM model. This mapping scheme took advantage of some of the special properties of the MasPar, although it was not fine-tuned for each individual routine. This mapping scheme was described in Chapter 10. (This approach has been used in simulating PRAM algorithms on various parallel architectures, e.g., see the section on simulating PRAM algorithms in [Lei92]. However, most of the previous results do not have any implementation details and provide no

performance data.) Using this mapping, we then coded each simple parallel primitive on the MasPar. While coding each primitive, we utilized the special properties of the MasPar to fine-tune our code. Since each parallel primitive is very easy to code, one would expect the fine-tuning step to be much simpler than the fine-tuning step of a complicated algorithm. We implemented a set of parallel graph algorithms without virtual processing by calling the parallel primitives we coded and routines provided in the system library as described in Chapter 11.

Due to the constraints imposed by the programming environment on the MasPar, the above implementation requires the size of the input to be no more than the number of available physical processors. However, the parallel primitives coded can be used with any number of processors by invoking Brent's scheduling principle [Bre74, KR90] to simulate several virtual processors on one physical processor. To do this, we extended our mapping scheme to handle the allocation and simulation of virtual processors. The extended mapping is described in Chapter 12.

Using our original code when no virtual processors are used as described in Chapter 11 as a blueprint and the extended mapping as a guideline, we transformed our code to handle the allocation of virtual processors. Since the MPL language does not support virtual processing, we had to implement our own scheme for virtual processing. To do this, we re-coded and fine-tuned the set of parallel primitives identified in Chapter 11 and several system library routines to handle the allocation of virtual processors efficiently. Then we implemented a set of parallel graph algorithms by calling these parallel primitives and system routines. The primitives and graph algorithms we implemented are described in Section 13.3.

13.3 Implementation of Parallel Graph Algorithms

In this section, we first describe the implementation of several data structures. Then we describe the parallel graph algorithms library that we have built.

13.3.1 Data Structures

Array and Linked List

Given the value of vpr, we mapped a global memory array used in a PRAM algorithm onto the MasPar by putting the ith element of the array into the ith VPE. Thus this element will be allocated in the (i mod vpr)th element of a local array on the ($\lfloor \frac{i}{vpr} \rfloor$)th physical processor. We mapped a linear linked list used in a PRAM algorithm by putting each element in the list into a different VPE. Pointers in PRAM were replaced by the ID's of VPE's.

Tree

We represented an edge in an undirected tree by two directed edges of opposite directions. A tree was represented by a list of directed edges. In implementing the tree data structure on the MasPar, we put one directed edge in one VPE with the requirement that the set of edges that are incoming to the same vertex have to be allocated on a consecutive segment of VPE's. Each of the two copies of an undirected edge kept a reverse pointer which pointed to the location of the other copy of the same edge. Using this representation, we can use the XNET connection to perform inter-processor communications needed for computing an Euler tour on a tree. Since computing an Euler tour is one of the most common subroutines on trees used by parallel graph algorithms, we saved time by using this mapping.

Undirected Graph

In our implementation without virtual processing as described in Chapter 11, a

general undirected graph was represented by a list of edges. Each edge had two copies with the two endpoints interchanged. We placed an edge on a MasPar PE with the requirement that the two copies of the edge have to be located on adjacent PE's. The reason for using this data structure was twofold. First, we wanted a tree to be represented by a list of edges such that edges incident on a node were allocated in a continuous segment of processors for the ease of finding an Euler tour in a tree. Representing an undirected edge by its two corresponding directed versions was consistent with the representation of a tree. Second, undirected graph algorithms often needed to perform operations on nodes based on information stored on the edges incident on them. Since an undirected edge has two endpoints, each edge had to perform operations on each of its two endpoints. Thus we needed two processors to handle one undirected edge. When virtual processing was involved, the natural candidate for our mapping was to allocate each copy of an edge on a different VPE.

Let m and n be the numbers of (undirected) edges and nodes in the input graph, respectively. Using the naive strategy for allocating undirected graphs described in the previous paragraph, we determined the value of vpr by computing the least power of 2 that is greater than or equal to $\left\lceil \frac{2m+2}{nproc} \right\rceil$. Edges were allocated among virtual processors with the ID's from 2 to $2m+1$. (For the easy of programming, we did not use the first two virtual processors for storing edges.) The ith node was allocated to the virtual processor with the ID i. Initially, we coded the routine for finding a spanning forest with virtual processing using this simple strategy.

In the case when m was much greater than n, this type of data allocation scheme was not balanced since only a small portion of the machine was performing computations related to nodes. The other drawback in using this

type of allocation came from the types of operations that were usually used in parallel graph algorithms. It is often the case that information related to edges incident on a node v had to be collected to produce data that will be stored in the processor that was allocated for v. In performing these operations, data will compete with each other to reach a small segment of processors that are physically connected to each other. The delay for this type of inter-processor communications was very large. In order to improve the performance of our code, we considered alternative strategies.

Dynamic Load Balancing One possible solution for the above problem was to compute different *vpr* values for nodes and for edges. However, for this we would have to revise our code for parallel primitives such that each primitive knew whether it was performing operations on edges or on nodes. Also, the code for our graph algorithms would have to be changed. This would result in a more complicated implementation. Instead of going through such a serious revision, we came up with the following simple method that did not require us to change other programs. We first computed the number of virtual processors per node to be $nfactor = \left\lfloor \frac{vpr \cdot nproc}{n} \right\rfloor$. We then allocated the ith node to the $(i \cdot nfactor)$th virtual processor. We chaned the node numbers referred to in each edge accordingly. This was done by multiplying $nfactor$ to every node number used in the edge list. We then performed all of our computations as if the number of nodes is $n \cdot nfactor$. (This is equivalent to adding $n \cdot (nfactor - 1)$ isolated vertices into the input graph.) After performing the computation, data related to nodes allocated in every other $nfactor$ virtual processors was collected. By performing simple preprocessing and post-processing, we evenly distributed all nodes and did not have to track the value of *vpr* during each operation. Our previous code for finding a spanning forest with virtual processing

could be used with minor modification.

Note that we could apply the same technique to several data structures used in our programs. For example, our graph algorithms often found a spanning forest in the input graph and obtained an Euler tour of each of the tree in the spanning forest. The total number of edges in the Euler tours of the forest was $2n - 2$. We could apply the same technique to achieve a better load balancing by evenly distributing tour edges among physical processors. Our graph algorithms also performed range minimum queries on an array of elements whose size was $2n - 2$. We could also use this technique to achieve a better load balancing by evenly distributing elements in the array among physical processors.

We tested the implementation of our parallel program for finding a spanning forest on graphs of three different edge densities: (1) dense graphs where $m = \frac{n^2}{4}$; (2) intermediate-density graphs where $m = n^{1.5}$; (3) sparse graphs where $m = \frac{3n}{2}$. Performance data is shown in Figure 13.1 for this problem with and without the usage of dynamic load balancing. Figure 13.1 shows that by using dynamic load balancing, our parallel program ran about 12 times faster than our parallel program without dynamic load balancing on dense graphs. On intermediate-density graphs, it was about 8 times faster. On sparse graphs, it was about 1.5 times faster. We expect this type of behavior as dynamic load balancing provides more help as the graph gets denser.

Compressed Data Structure A major goal of our implementation was to run inputs whose sizes are as large as possible. Since we have a limited amount of memory space per physical processor, we wanted to minimize the amount of space used by each edge without paying too much overhead in computation. It turns out that except for the case of representing a tree for finding an Euler

tour, we can easily simulate the effect of having two processors handling one undirected edge by performing computations twice, one from each direction. Thus our program only allocated one processor to handle each edge. A side effect of this allocation scheme is that we had to write an expansion routine to convert this compressed representation into the tree format if we needed to build an Euler tour. In summary, our program first allocated vpr virtual processors per physical processor, where vpr is the least power of 2 that is greater than or equal to $\left\lceil \frac{m}{nproc} \right\rceil$. In the case when $2n > vpr \cdot nproc$ and a spanning tree format was needed, we doubled the value of vpr and called the expansion routine to transform the compressed data structure into the normal graph representation.

The performance data for running our program with and without the usage of compressed data structures for graphs to find a spanning forest are illustrated in Figure 13.2. Figure 13.2 shows that our program ran at about the same speed with or without the usage of compressed data structures on dense graphs and intermediate-density graphs. Note that by using compressed data structures, we could double the size of the largest graph we could handle. For sparse graphs, we had to pay a little overhead in using compressed data structures. Since the overhead was small, we decided to use compressed data structures for graphs though the code became a bit longer. Thus when allocating vpr virtual processors to each physical processor, we could run our programs on graphs with $vpr \cdot nproc$ nodes and $vpr \cdot nproc$ edges if we did not require the usage of a spanning forest in the program. We could run programs on graphs with $\frac{vpr \cdot nproc}{2}$ nodes and $vpr \cdot nproc$ edges if we had to use a spanning forest representation during the computation. Without the usage of compressed data structures, we could only run our programs on graphs with half the number of edges.

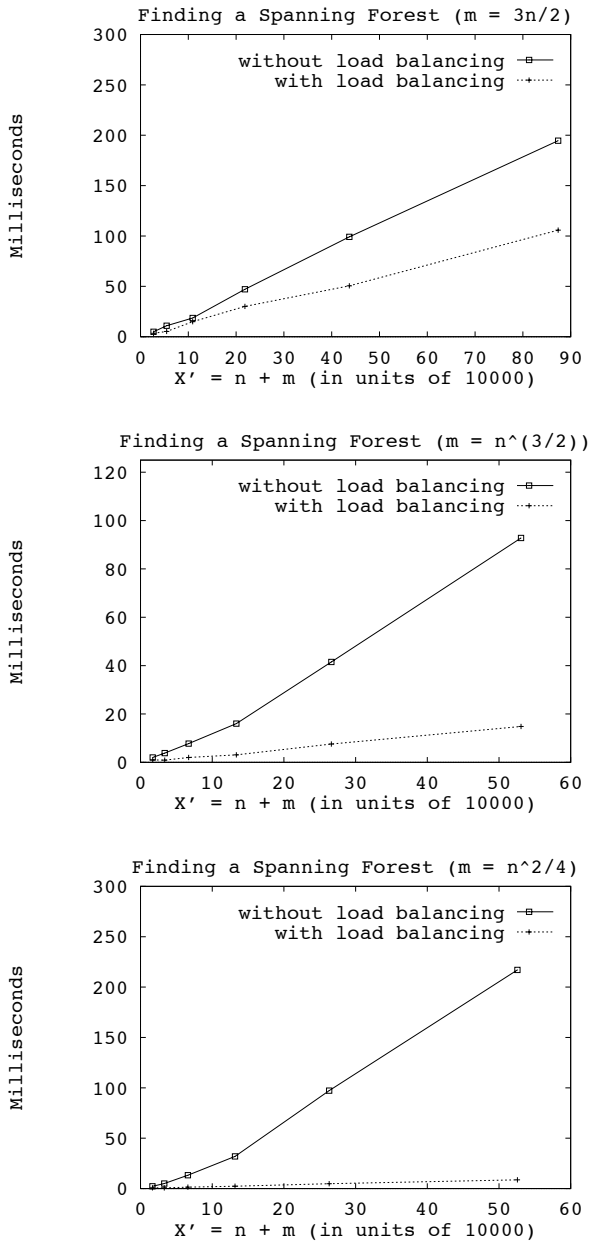


Figure 13.1: Illustrating the performance data for our parallel program for finding a spanning forest in graphs with and without dynamic load balancing.

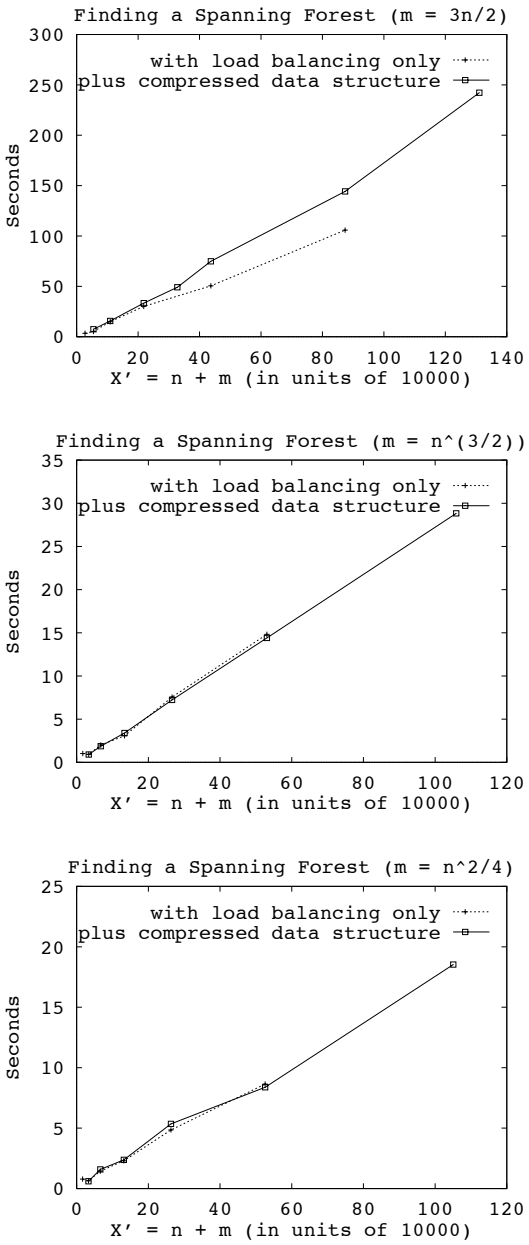


Figure 13.2: Illustrating the performance data for our parallel program for finding connected components in graphs with and without the use of compressed data structure.

13.3.2 The Parallel Graph Algorithms Library

To build our parallel graph algorithms library, we first wrote a kernel that includes all of the commonly used subroutines for designing parallel graph algorithms. Then we built our graph application programs by calling routines in the kernel and routines provided in the system library. The structure of the whole library is shown in Figure 13.3.

Routines in the System Library and the Kernel We briefly describe the routines in the kernel. All of these routines are based on PRAM algorithms that run in $O(\log n)$ time for an input of size n. Although some of them are not theoretically optimal algorithms in that they perform $\Theta(n \log n)$ work, they are within an $O(\log n)$ factor of optimality, and they are very simple. These routines are as follows. (1) List ranking [KR90]. (2) Rotation. This routine rotates the data stored in a processor with ID i to the processor with ID $(i + d) \bmod p$, where d is an input to the routine and p is the number of PE's in the system. (3) Segmented rotation. We store data in each processor and partition the set of processors into sequences of consecutive segments. This routine rotates the data stored in each processor within each segment. Data within each segment are rotated in a way similar to the rotation routine described in (2). (4) Range minimum [TV85]. (5) Euler tour construction [TV85]. (6) Preorder numbering [TV85]. (7) Least common ancestor [TV85].

When implementing the above routines in the kernel without virtual processing as described in Chapter 11, we also used the following routines that are provided in the system library. (1) Sorting. (2) Prefix sums. (3) Inter-processor communication. (4) Data combining. In our implementation of parallel algorithm with virtual processing, we also used the above routines.

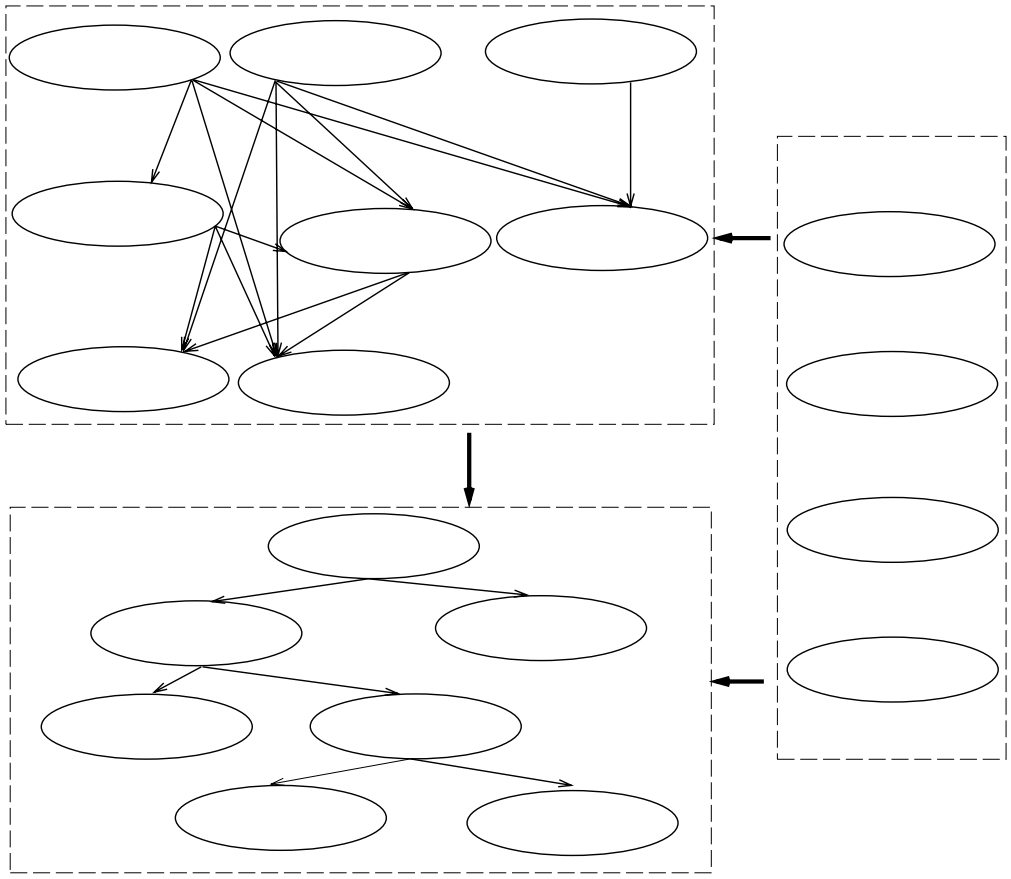


Figure 13.3: The structure of the routines we built for the parallel graph algorithms library. The kernel of the library will be used by the application routines. In our coding, we also use routines provided in the system library. An arrow from one node to another node means the routine at the tail of the arrow (upper) is used by the routine at the head of the arrow (lower).

Since the MasPar does not provide virtual processing for these system routines, we coded and fine-tuned all of these routines with virtual processing except sorting. For sorting with virtual processing, we used the package developed in [PS90]. Since the sorting package in [PS90] can only handle the case when the number of virtual processors simulated by each physical processor is a power of two, our code inherited the same restriction. The implementation of routines in the system library and the kernel is described in Chapter 12.

Graph Application Routines We implemented parallel algorithms for the following problems using the above kernel. (1) Spanning forest [AS87]. (2) Minimum cost spanning forest [AS87]. (3) Cut edges [Ram93]. (4) Ear decomposition of a two-edge-connected undirected graph [Ram93]. (5) Open ear decomposition of a biconnected undirected graph [Ram93] (6) Strong orientation of a two-edge-connected undirected graph [Ram93].

13.4 Performance Analysis

We tested our code by generating test graphs and measuring the performance of the code on these test graphs. In addition to testing our parallel code for the problems listed in Section 13.3.2, we also took the implementation of their corresponding sequential algorithms described in Chapter 11 and tested them on large inputs using SUN SPARC workstations. The corresponding sets of performance data were compared and studied. Note that a MasPar MP-1 PE is about 200 times slower than a SUN SPARC II and about 230 times slower than a SUN SPARC 10/41. Thus it is to be expected that the performance of our sequential programs will be faster than our parallel programs in some cases, though the speed-up of our parallel implementation was quite good, given the parameters of the MasPar MP-1. One noteworthy feature of our parallel imple-

mentation is that it could handle inputs whose sizes are much larger than the the sizes of the input that can be handled by our sequential implementation.

The organization of this section is as follows. We first describe the method we used in generating test graphs. Then we describe the way we tested our programs and the curve-fitting scheme we performed on the sets of performance data. Finally, we analyze the performance data.

13.4.1 Generation of Test Graphs

We tested our programs using graphs of three different edge densities as described in Section 13.3.1 and Chapter 11. For testing the code for finding a spanning forest and a minimum spanning forest, we generated test graphs from the class of random graphs $G_{n,m}$ as described in Chapter 11. In addition, a random cost in the range from 0 to 99,999 (with repetition) on each edge, instead of from 0 to 999 as used in Chapter 11, was generated for testing the routine for finding a minimum spanning forest. Test graphs with a given edge density, a given size, and a given property (e.g., biconnectivity) were generated using a similar method as described in Section 11.4.2.

13.4.2 Testing Scheme

For each size and sparsity, we generated four different test graphs. We ran each program on each test graph for 10 iterations and recorded the average of the 40 trials. The results are plotted in Figures 13.4 – 13.8.

We had access to a MasPar MP-1 machine with 16,384 processors and 32 kilobytes of available memory per processor. (The other 32 kilobytes of memory in each processor was not available to us.) We were able to test all of our programs except the one for finding an open ear decomposition for

the value of vpr up to 64. We were only able to run our parallel program for finding an open ear decomposition for the value of $vpr = 32$. Note that for testing dense graphs and intermediate-density graphs, we could run programs on graphs with $m = vpr \cdot 16{,}384$. For testing sparse graphs ($m = \frac{3}{2}n$), we could only run inputs with $m = \frac{3}{4} \cdot vpr \cdot 16{,}384$ since a tree data structure is required in the computation and we needed $2n = \frac{4}{3}m$ virtual processors to represent it. Our parallel programs for finding a spanning forest and for finding a minimum spanning forest used 24 kilobytes per physical processor when the value of vpr was 64. The rest of the programs used 32 kilobytes for the largest inputs that we have tested. We spent about 2 months to obtain all of our performance data.

We ran the set of sequential algorithms implemented in Chapter 11 on a SUN SPARC 10/41 machine with 32 megabytes of memory and about 80 megabytes of swapping space on input sizes greater than 16,384. We tested the sequential programs on larger and larger inputs until either the programs complained that the usage of the memory is too much or we waited more than 1 day while there was only one active job running on the machine. For sparse graphs, our sequential programs ran out of available memory before we could obtain performance data that was worse than the corresponding parallel program. However, on dense graphs and intermediate-density graphs, our parallel algorithms run much faster (in real CPU time) than their sequential counterparts. The likely reason is that we use a depth-first search in our sequential programs, which is a recursive program whose depth of recursion could be as large as the number of nodes in the graph.

Our sequential programs were implemented with the help of the graph package NETPAD [DMM92] developed in Bellcore as described in Chapter 11.

NETPAD uses a lot of extra memory in creating a standard graph data structure. Thus we might save space by coding the sequential algorithms from scratch. We also note that the turn-around time (wall-clock time) for each of our sequential programs was very large when we used more than 80% of the main memory even if the system had only one job active, though our time measurement routine would report only a small fraction of the turn-around time. For example, for finding a minimum spanning forest sequentially on graphs with 300,000 edges, the time measurement routine reported a total usage of 110 seconds for 10 iteration of our program. However, the turn-around time was about 20 hours. We conjecture the reason might be that the architecture of SPARC 10/41 handles swapping poorly. We were unable to find better routines for measuring the performance of our sequential programs on the SPARC 10/41. For 5 of our 6 parallel programs, we were able to obtain sequential performance data that was worse than their parallel counterpart by testing large inputs. For the sequential algorithm for finding a minimum spanning forest, the turn-around time was too long when the input graph had more than 300,000 edges. As a result, we did not obtain further performance data for finding a minimum spanning forest such that we could observe the place where the sequential performance was worse than the parallel performance as shown in other programs. Overall, we spent more than 2 months in getting all of the performance data for our sequential programs. For all problems, we could handle input whose sizes are 4 to 5 times larger using our parallel code.

13.4.3 Least-Squares Curve Fitting

We applied the least-squares fit package in Mathematica [Wol88] to the data we obtained. We used the following method to find the fitted curves for our performance data. We first derived the theoretical asymptotical running

time for our parallel program. For example, our code for finding a spanning forest in a graph with n nodes and m edges runs in $O(\frac{n}{p} \cdot \log^3 n)$ time using p processors since we used an $O(\log^2 n)$ time bitonic sorting routine in implementing global concurrent write operations. We first used Mathematica to find coefficients c_0, c_1, c_2, c_3 and c_4 such that the function $c_0 + c_1 \cdot x + c_2 \cdot x \cdot \log x + c_3 \cdot x \cdot \log^2 x + c_4 \cdot x \cdot \log^3 x$ best fit the set of experimental data that we obtained with virtual processing.

If any of the coefficients was negative, we forced the negative coefficient c_i with the largest integer i to be zero and perform the fitting once again. We iterated this process until all coefficients were not negative. We also performed the least-squares fit for performance data of the sequential programs when the amount of memory used in the program was within the capacity of the main memory.

To test the goodness of the curve we obtained, we computed the *average error* as the square root of $\frac{1}{k} \cdot \sum_{i=1}^{k} (\frac{y_i - f(x_i)}{f(x_i)})^2$, where k is the number of experimental data points, f is the function that describes the fitted curve and y_i is the experimental value when the input size is x_i.

13.4.4 Analysis

In Section 13.4.4 through Section 13.4.4, we present the performance of our code for each of our six graph problems. In the following, x' is the size of the input in units of 10,000. In interpreting the following data, note that we present the fitting curves in terms of x' when virtual processors are used. There is a further compression by a factor of 2 due to the compressed data structure when virtual processors are used. The function value of each fitted curve is the running time in seconds.

Finding a Spanning Forest For the parallel implementation, we modified the CRCW PRAM algorithm in [AS87] for finding connected components to find a spanning forest of the input graph. The original algorithm partitions the set of vertices into a set of disjoint sets such that vertices in each set are in the same connected component. Initially, the algorithm puts a vertex in each set. During the execution, the algorithm merges two sets of vertices if they are detected to be in the same connected component. Our program selects an edge connecting a vertex in one set to a vertex in the other set while merging these two disjoint vertex sets. The sequential algorithm that we implemented is the simple linear time depth-first search algorithm.

The performance data with virtual processing is shown in Figure13.4. The fitted curves for the parallel performance data with virtual processing are $0.0014x' \log^3 x' + 1.32x' + 1.41$ (with 7.1% average error), $0.0000091x' \log^3 x' + 0.27x' + 0.028$ (with 3.2% average error), and $0.000074x' \log^3 x' + 0.15x' + 0.45$ (with 9.1% average error) for sparse graphs, intermediate-density graphs and dense graphs respectively. The corresponding fitted curves for the sequential performance data when the data is within the main memory are $0.17x'$, $0.17x'$, and $0.15x'$.

Finding a Minimum Spanning Forest For the parallel implementation, we modified the algorithm in [AS87] for finding connected components to find a minimum cost spanning forest for the input graph. This algorithm also partitions the graph into disjoint sets of vertices. In addition, for each current set of vertices, we compute a minimum edge with exactly one endpoint in the set using the concurrent write operation. This edge determines which other set of vertices is to be merged with its set. Once the merge is completed, the edge that caused the merging is marked as one of the edges in the minimum cost

spanning forest. For sequential implementation, we implemented the $O(n + m \log n)$-time Kruskal's algorithm [Tar83] for finding a minimum cost spanning forest. Although faster algorithms are known for this problem, we implemented Kruskal's algorithm for its simplicity.

The performance data with virtual processing is shown in Figure 13.5. The fitted curves for the parallel performance data with virtual processing are $0.0015x' \log^3 x' + 0.78x' + 2.46$ (with 14% average error), $0.00037x' \log^3 x' + 0.68x' + 0.11$ (with $0.00021x' \log^3 x' + 0.67x'$ (with 7.2% average error) for sparse graphs, intermediate-density graphs and dense graphs respectively. The corresponding fitted curves for the sequential performance data when the data is within the main memory are $0.1x' \log x' + 5.44$, $0.056x' \log x' + 1.78$, and $0.049x' \log x' + 1.22$.

Finding All Cut Edges Our parallel implementation is based on [Ram93]. We first obtained a rooted spanning tree T for the input graph G. (The current version of the program requires G to be connected.) A cut edge is a tree edge (u, v), where u is the parent of v and there is no non-tree edge (x, y) in G such that either x or y is a descendant of v or equal to v and the least common ancestor of x and y is a proper ancestor of v. This can be determined by using the Euler tour technique and the range minimum queries. For sequential implementation, we used a linear time algorithm for finding all cut edges in the graph based on depth-first search [Ram93].

The performance data with virtual processing is shown in Figure 13.7. The fitted curves for the parallel performance data with virtual processing are $0.0019x' \log^3 x' + 1.44x' + 1.59$ (with 10.2% average error), $0.00018x' \log^3 x' + 0.61x' + 0.27$ (with 2.9% average error), and $0.00043x' \log^3 x' + 0.49x' + 0.73$ (with 3.7% average error) for sparse graphs, intermediate-density graphs and

dense graphs respectively. The corresponding fitted curves for the sequential performance data when the data is within the main memory are $0.22x'$, $0.18x'$, and $0.16x'$.

Finding an Ear Decomposition For the parallel implementation, we used the PRAM parallel algorithm in [Ram93] for finding an ear decomposition of a 2-edge-connected graph by calling the sorting routine, routines in the kernel and the routine for finding a spanning forest. For sequential implementation, we used a linear time algorithm for finding an ear decomposition based on depth-first search [Ram93].

The fitted curves for the parallel performance data with virtual processing are $0.0014x' \log^3 x' + 1.43x' + 0.36$ (with 5.0% average error), $0.00036x' \log^3 x' + 0.33x' + 0.2$ (with 11.0% average error), and $0.00064x' \log^3 x' + 0.21x' + 0.91$ (with 5.9% average error) for sparse graphs, intermediate-density graphs and dense graphs respectively. The corresponding fitted curves for the sequential performance data when the data is within the main memory are $0.57x'$, $0.72x'$, and $0.68x'$.

Finding an Open Ear Decomposition For the parallel implementation, we used the PRAM algorithm in [Ram93] for finding an open ear decomposition. This routine is obtained by modifying the ear decomposition algorithm mentioned in the previous section. The sequential ear decomposition algorithm mentioned in the previous section [Ram93] also finds an open ear decomposition of a biconnected graph.

The performance data with virtual processing is shown in Figure 13.9. The fitted curves for the parallel performance data with virtual processing are $0.0017x' \log^3 x' + 1.59x' + 0.24$ (with 13.9% average error), $0.0014x' \log^3 x' +$

$1.03x' + 0.36$ (with 9.0% average error), and $0.0024x' \log^3 x' + 0.057x' \log x' + 0.96x' + 0.5$ (with 7.1% average error) for sparse graphs, intermediate-density graphs and dense graphs respectively. Note that the sequential performance data for finding an open ear decomposition is the same as the sequential performance data for finding an ear decomposition. We will not restate them here.

Finding a Strong Orientation For the parallel implementation, we first obtained an ear decomposition for the input graph. Then we directed the edges of each ear so that each ear forms a directed path or a directed cycle. Observe that the ear decomposition algorithm first obtained rooted spanning tree T. The edges in an ear are of the form (v_1, v_2), (v_2, v_3), ... , (v_{k-1}, v_k), (v_k, u_r), (u_r, u_{r-1}), (u_{r-1}, u_{r-2}), ... , (u_2, u_1), where (v_i, v_{i+1}) is a tree edge and v_i is the parent of v_{i+1} in T, for $1 \leq i < k$; (u_{i+1}, u_i) is a tree edge and u_{i+1} is the parent of u_i in T, for $1 < i \leq r$; (v_k, u_r) is a non-tree edge. Thus we directed every non-tree edge (u, v) from u to v where u has a smaller ID than that of v. Then we assigned directions to tree edges in such a way that the edges in an ear formed a directed path or directed cycle and the first two ears together formed a directed cycle. For sequential implementation, we used a linear time algorithm for finding a strong orientation based on a recursive version of depth-first search [Tar72].

The performance data with virtual processing is shown in Figure 13.8. The fitted curves for the parallel performance data with virtual processing are $0.0021x' \log^3 x' + 1.39x' + 1.78$ (with 5.8% average error), $0.00012x' \log^3 x' + 0.42x' + 0.032$ (with 2.6% average error), and $0.00058x' \log^3 x' + 0.23x' + 0.89$ (with 5.1% average error). for sparse graphs, intermediate-density graphs and dense graphs respectively. The corresponding fitted curves for the sequential

	$m = 3n/2$		$m = n^{3/2}$		$m = n^2/4$	
	no vpr m = 8,191 (seconds)	$vpr = 16$ m = 262,142 (seconds)	no vpr m = 8,191 (seconds)	$vpr = 16$ m = 262,142 (seconds)	no vpr m = 8,191 (seconds)	$vpr = 16$ m = 262,142 (seconds)
Spanning Forest	1.01	74.86	0.41	7.23	0.39	5.35
Minimum Spanning Forest	1.05	51.97	0.73	18.58	0.70	19.69
All Cut Edges	1.17	83.36	0.61	17.92	0.57	15.85
Ear Decomposition	1.19	72.54	0.60	11.32	0.58	8.71
Open Ear Decomposition	1.47	90.35	1.11	33.69	0.94	33.50
Strong Orientation	1.20	75.35	0.63	11.61	0.60	9.06

Table 13.1: Performance data for our parallel programs with and without virtual processing. The data for parallel programs without virtual processing is from Chapter 11.

performance data when the data is within the main memory are $0.31x'$, $0.25x'$, and $0.22x'$.

13.4.5 Overhead for Implementing Virtual Processors

We compared the amount of time used by our parallel programs with and without virtual processing. The performance data is shown in Table 13.1. Note that we ran 5 of our 6 programs for the value of vpr up to 64 using no more than half of the available memory in the system. The one program that we could run only up to the value of $vpr = 32$, was the open ear decomposition routine. Also, when $vpr = 32$, our code for open ear decomposition could not handle inputs of size 32·16,384 on sparse graphs. (See Section 13.4.2 for details.) Hence in Table 13.1, we use $vpr = 16$ to show the performance of our parallel code when the virtual processors simulated in each physical processor were all active. The performance data without virtual processing is from Chapter 11. Our implementation of parallel algorithms with virtual processing had excellent speed-ups on dense graphs and intermediate-density graphs in relation to the implementation without virtual processing. For example, for finding an ear decomposition on dense graphs, we used 15 times more CPU time with virtual

processing while handling graphs that were 32 times larger. For sparse graphs, the overhead was fairly large. The reason might be that for sparse graphs, using virtual processors increased the degree of concurrency when concurrent read or write is used. Since we could not offset it by the use of dynamic load balancing, our implementation had a big overhead on sparse graphs. We also note that the overhead for implementing the open ear decomposition algorithms is about twice as large as the overhead for implementing other algorithms.

13.5 Concluding Remarks

We have implemented a set of parallel algorithms for undirected graphs on the MasPar MP-1 to handle sizes of the input that are larger than the number of available physical processors. We tested our parallel programs on inputs whose sizes were up to 64 times larger than the number of physical processors and compared their performance with corresponding sequential programs. Note that by using the full configuration of the current machine, we can simulate up to 128 virtual processors per physical processor. However, sharing the machine with other users limited us to use only half of the available memory in each processor. Thus if the full machine had been available, we could have run our programs on graphs with one million nodes and two million edges.

We note the following observations.

- By using the high-level structure of the PRAM algorithms as building blocks, our coding and debugging effort was relatively small. We wrote more than 12,000 lines of parallel code for the set of parallel graph algorithms that we implemented with virtual processing. All of the work reported here (include testing) was done within one year. Note that 4,000

lines of parallel code were written in 12 weeks for our implementation described in Chapter 11 for the same set of parallel programs without virtual processing. We consider our strategy for implementing parallel graph algorithms to be promising.

- We examined the variation in data we obtained on the four different test graphs of a given size and a given edge density. Most of the data points ($> 90\%$) were within 7% of their average values. Less than 5% of the data points were more than 15% away from their average values.

- We compared the experimental data points with the computed points on the fitted curves. The average error was about 10% for all data sets with virtual processing. Our fitted curves fit quite well on the experimental performance data. The fitted curves showed that the dominant term in our parallel code was $\log^3 n$. We conjecture the reason might be that our graph algorithms usually compute by performing $O(\log n)$ iterations and if each iteration takes $\Theta(\log^2 n)$ time, the overall time complexity is $O(\log^3 n)$.

- Sequential implementations usually performed badly when a fraction of their data were placed out of the main memory. Note that our sequential programs used extra memory because we used NETPAD. Thus the biggest size inputs that one can run on a SPARC 10/41 would be somewhat larger than what we have shown here if a more efficient coding of the graph data structure is used. It is also possible to run larger inputs on our sequential programs if we use a machine with larger main memory. The current machine we used has 32 megabytes of main memory, while machines with up to 1 gigabytes of main memory are currently available.

However, one would expect the total memory in a parallel computer to be much larger than the size of any sequential machine once the market demands the existance of such machines.

- Although each MasPar MP-1 PE is much slower than the SPARC workstation, we found that in most of the cases, parallel programs in fact runs faster in real time compared to sequential programs. In particular, our parallel programs were much faster on dense graphs and intermediate-density graphs than on sparse graphs. We traced our parallel program for finding a spanning forest and noticed that by using our dynamic load balancing technique, the performance of a concurrent read or write was not too bad on a dense graph compared to the performance of the same operations on a sparse graph. Recall that our algorithm for finding a spanning forest obtained a spanning forest by repeatedly growing forests in parallel in a loop until the size of any tree in the current forest could not be expended. For dense graphs, the parallel algorithms terminated in fewer iterations than on sparse graphs. Thus the running time was much smaller on dense graphs than on sparse graphs. Our parallel code can also handle much larger inputs than our sequential code.

- We found that our sequential program for finding a spanning forest used about 45 megabytes of memory on the largest inputs. Our parallel program used no more than 24 kilobytes of memory per physical processor on inputs whose sizes were more than 4 times larger than the size of the largest inputs for the sequential program. Since there are 16,384 physical processors in the MasPar MP-1, the total memory used in our parallel program was no more than 384 megabytes. Hence we used about 8 times more memory in our parallel programs while we could run inputs

whose sizes were 4 times larger than the largest input size on the SPARC 10/41. In most cases, when testing the largest size inputs, our parallel code ran faster than their sequential counterparts on dense graphs and intermediate-density graphs even when the input size was 4 times larger.

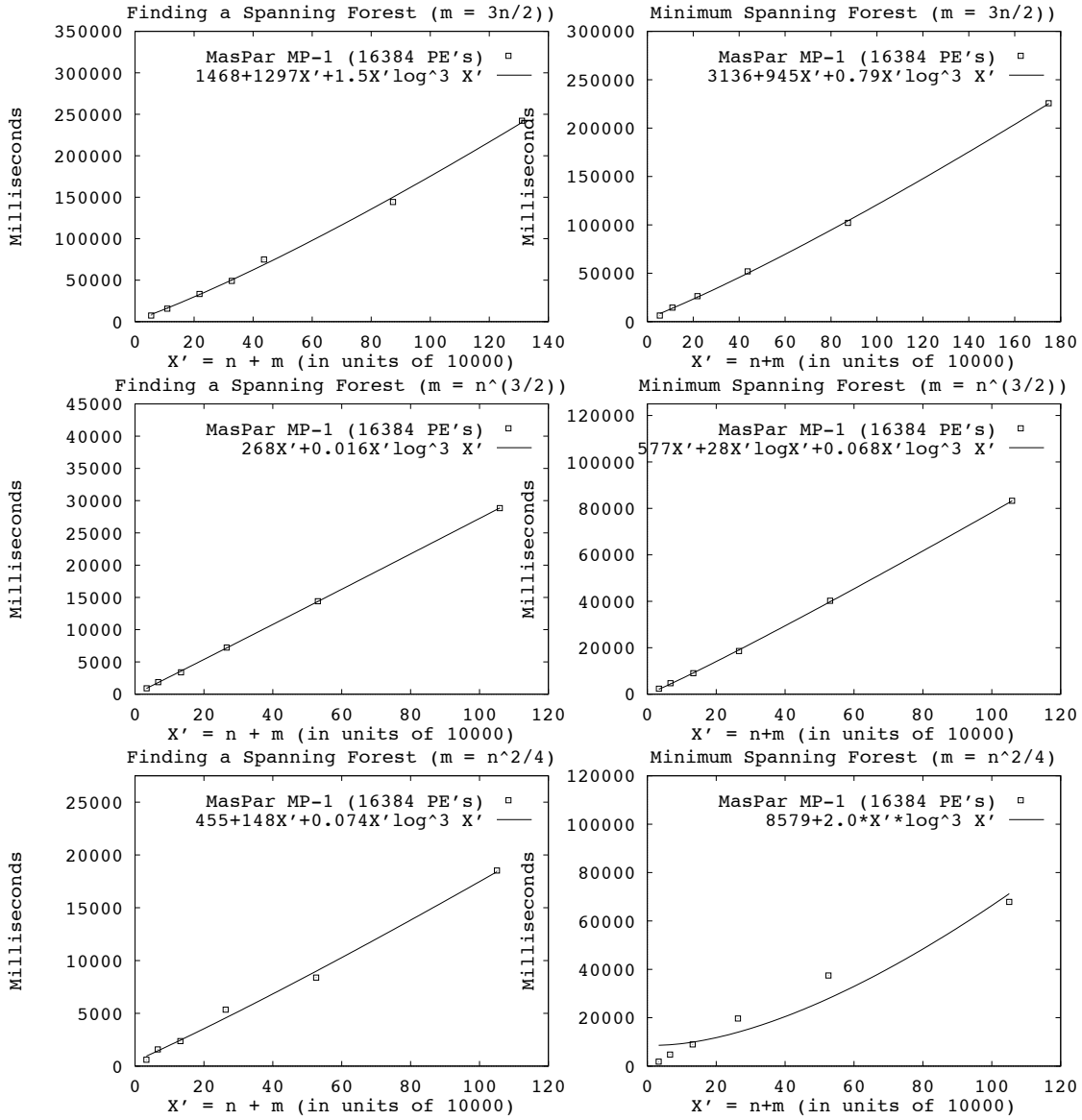


Figure 13.4: Relative performance of the sequential program on a SPARC 10/41 workstation and the parallel program on the MP-1 for finding a spanning forest with virtual processing. The least-squares-fit curves for the sequential performance data when $< 80\%$ of the main memory are used are $0.17x$, $0.17x$, and $0.15x$, respectively, from the top to the bottom.

Figure 13.5: Relative performance of the sequential program on a SPARC 10/41 workstation and the parallel program on the MP-1 for finding a MSF with virtual processing. The least-squares-fit curves for the sequential performance data when $< 80\%$ of the main memory are used are $5.4 + 0.2x \log x$, $1.8 + 0.1x \log x$, and $1.2 + 0.1x \log x$, respectively.

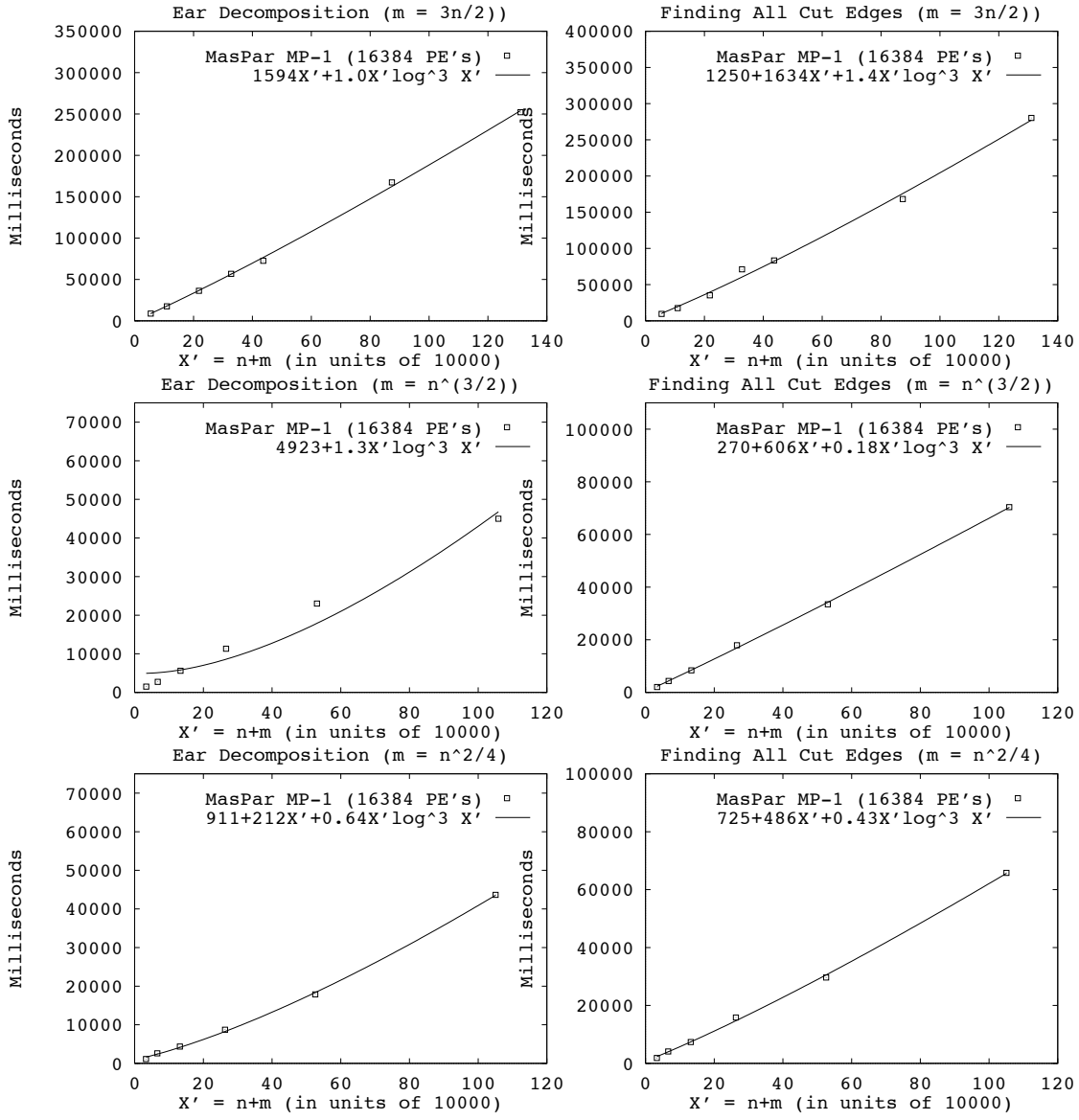


Figure 13.6: Relative performance of the sequential program on a SPARC 10/41 workstation and the parallel program on the MP-1 for finding an ear decomposition with virtual processing. The least-squares-fit curves for the sequential performance data when $< 80\%$ of the main memory are used are $0.57x$, $0.72x$, and $0.68x$, respectively, from the top to the bottom.

Figure 13.7: Relative performance of the sequential program on a SPARC 10/41 workstation and the parallel program on the MP-1 for finding all cut edges with virtual processing. The least-squares-fit curves for the sequential performance data when $< 80\%$ of the main memory are used are $0.22x$, $0.18x$, and $0.16x$, respectively, from the top to the bottom.

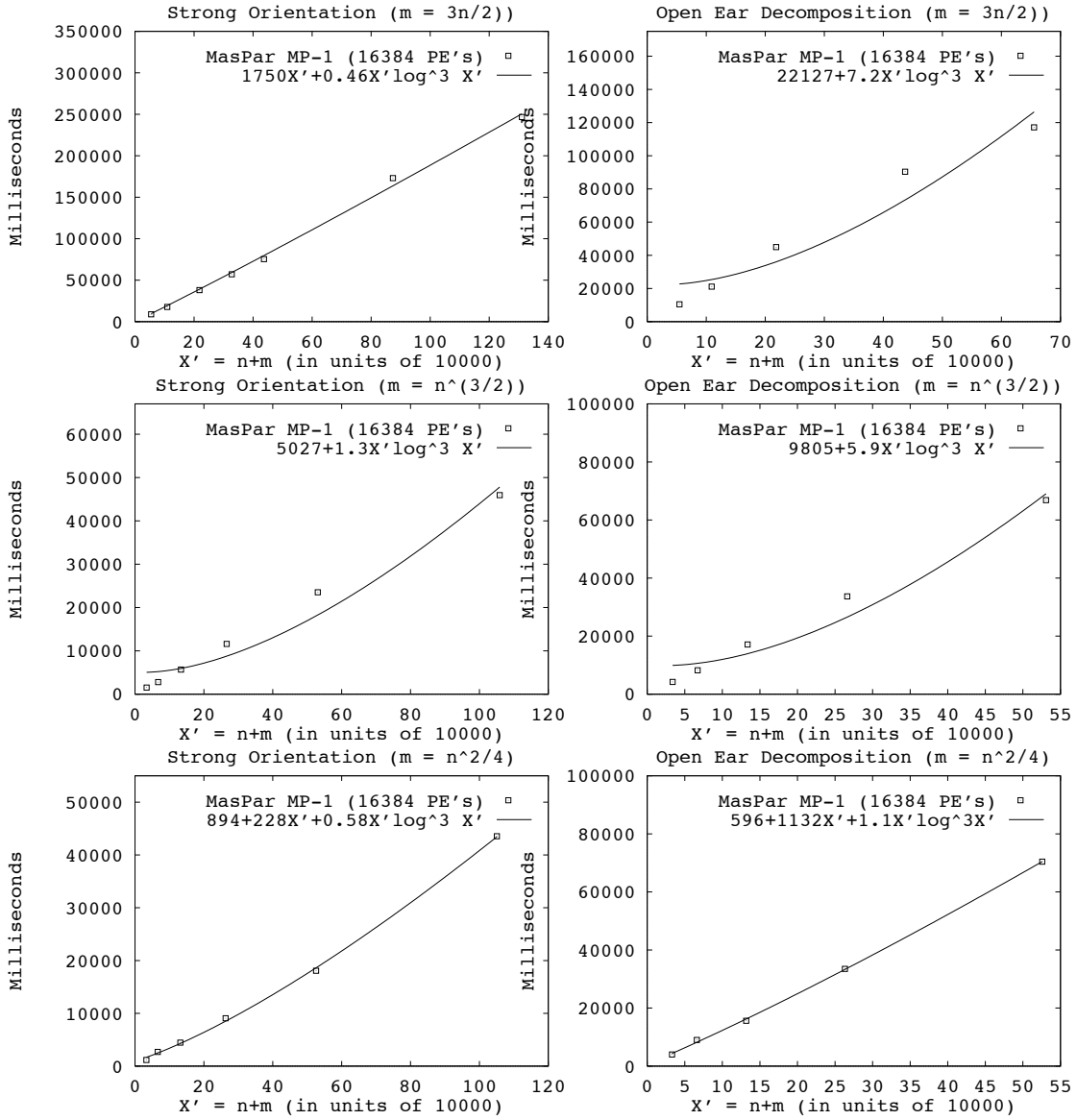


Figure 13.8: Relative performance of the sequential program on a SPARC 10/41 workstation and the parallel program on the MP-1 for finding a strong orientation with virtual processing. The least-squares-fit curves for the sequential performance data when $< 80\%$ of the main memory are used are $0.31x$, $0.25x$, and $0.22x$, respectively, from the top to the bottom.

Figure 13.9: Relative performance of the sequential program on a SPARC 10/41 workstation and the parallel program on the MP-1 for finding an open ear decomposition with virtual processing. The least-squares-fit curves for the sequential performance data when $< 80\%$ of the main memory are used are $0.57x$, $0.72x$, and $0.68x$, respectively, from the top to the bottom.

Chapter 14

Summary and Future Work

14.1 Summary

In Part II, we have presented our work on implementing efficient PRAM-based algorithms for undirected graphs on a massively parallel SIMD computer, the MasPar MP-1. Our implementation project is divided into two stages, which we summerize below.

In the first stage, for ease of programming, we assume that the input size is no more than the number of physical processors. We observed that PRAM algorithms for complex undirected graph problems are usually derived by calling fundamental parallel primitives. Thus our implementation strategy is to first code a set of commonly used routines (the kernel) that are frequently used in solving undirected graph problems. Building on the top of the kernel, we implemented complex PRAM undirected graph algorithms. The code for the graph algorithms we have implemented consist of a sequence of kernel calls and simple data manipulations in between calls. To achieve good performance for our parallel implementation, we fine-tuned each routine in the kernel according the target machine properties (e.g., the MasPar MP-1). Since each primitive is very simple, the fine-tuning process for the kernel was easy.

In the second stage, we extended our implementation to handle the allocation of virtual processors. The programming language MPL that we used does not support virtual processing, though is very efficient. To implement virtual processing, we used the non-virtual processing code written in the first

stage as a blue print to derive our revised code with virtual processing. We first provided simple translating rules for handling data allocation. Using simple translating rules, we also rewrote our high-level non-virtual processing code for the graph algorithms line by line. Furthermore, we fine-tuned each routine in the kernel using different algorithms and picked the best implementation for each routine. We also implemented system routines with virtual processing. By doing the above, we have implemented a set of efficient parallel algorithms for undirected graphs with virtual processing.

Our experience as obtained from work done in Part II shows that PRAM algorithms provide a useful way of decomposing complex problem on undirected graphs. By using our approach, it is easy to code complex algorithms and the performance of implemented algorithms track theoretical prediction quite well.

14.2 Future Work

There are several avenues for future work. We list some of them:

- With a better understanding of the MPL language and the MasPar architecture, we could fine tune our programs to run faster. Some of the things that could be done include better utilization of registers in each PE, using faster I/O interface between the PE's and the MasPar file system and finding the trade-off between performing computations on the ACU and on the PE.

- Since we have implemented most of the commonly used routines for implementing PRAM undirected graph algorithms, we expect that it will be fairly easy to implement other graph algorithms; for example,

the routines for testing 3-edge-connectivity, triconnectivity and planarity [Ram93, RR89], since we have already implemented most of the basic subroutines for these problems.

- The lack of a good graph manipulation package like NETPAD for handling large graphs makes it difficult to debug our programs. In Chapter 11, NETPAD was able to help the debugging and testing of our parallel implementation after we built an interface to use it on the MasPar. In our current implementation, the sizes of the graphs became too large for NETPAD to handle. Work should be done for graph manipulation (especially visualization) packages on large graphs.

- Our current implementation requires that vpr, the number of virtual processors simulated by each physical processor, be a power of 2 because of a bitonic sorting package [PS90] that we used. We would like to replace this sorting package by a sorting routine that can simulate any number of virtual processors per physical processor.

- The usage of randomization speeded up our parallel implementation of the list ranking algorithm, though it used extra memory space. In general, randomized algorithms are usually very simple and easy to code. We would like to exploit the usage of randomization in our fine-tuning of parallel primitives. Work should also be done to improve the system pseudo-random number generator such that it will generate good pseudo-random numbers even if only a few processors are active.

- Note that the code rewriting rules that we gave in Chapter 12 are very structured and regular. We could automate the code-rewriting process by further formalizing the rules. Thus one can enjoy the convenience of

programming in the high-level language without paying too much effort in coding virtual processing.

- We note that our current implementation with virtual processing has a large overhead on sparse graphs. More work has to be done to improve the running time on graphs that are very sparse.

Appendix A

Proofs

Proof of Lemma 5.4.2:

We prove the lemma by contradiction. Let $|\Im_1| = \{c_1, c_2, \ldots, c_k\}$, $k \geq 3$. Let s_1 and s_2 be two Tutte pairs in G such that s_i contains c_i, $i \in \{1, 2\}$. Let $s_1 = (a_1, a_2)$ and let $s_2 = (a_3, a_4)$. Let $a_1 = c_1$ and let $a_3 = c_2$. Let $\mathcal{A} = \{a_i \mid 1 \leq i \leq 4\}$. We know that

$$\sum_{i=1}^{2} sd(s_i) = \sum_{i=1}^{2} d_3(s_i) + \sum_{i=1}^{4} d_2(a_i) - 6. \tag{A.1}$$

We then find out the possible lower bound value for $w(G)$ in the following cases.

Case 1: s_1 and s_2 share a common cutpoint and they are in different 2-blocks.

We assume that $a_2 = a_4$. There are at least $d_2(a_1) - 1$ a_1-components for s_1, $d_2(a_3) - 1$ a_3-components for s_2, $d_2(a_2) - 2$ a_2-components for s_1 which are also a_4-components for s_2, $d_3(s_1)$ s_1-components and $d_3(s_2)$ s_2-components. Those components share only vertices in $\{a_1, a_2, a_3\}$. From Corollary 5.4.15, we know that

$$w(G) \geq 2\sum_{i=1}^{3} d_2(a_i) - 8 + \sum_{i=1}^{2} d_3(s_i).$$

Case 2: s_1 and s_2 share a common vertex that is not a cutpoint or they share a common cutpoint and s_1 and s_2 are in the same 2-block .

In this case, s_1 and s_2 are in the same 2-block. We assume that $a_2 = a_4$. There are at least $d_2(a_1) - 1$ a_1-components for s_1, $d_2(a_3) - 1$ a_3-components for s_2, $d_2(a_2) - 1$ a_2-components for s_1 which are also a_4-components for s_2,

$d_3(s_1)-1$ s_1-components that do not contain any s_2-component and $d_3(s_2)$ s_2-components that do not contain any s_1-component. Those components share only vertices in $\mathcal{A}$. Thus

$$w(G) \geq 2\sum_{i=1}^{3} d_2(a_i) - 6 + \sum_{i=1}^{2} d_3(s_i) - 2.$$

Case 3: $|\mathcal{A}| = 4$ and s_1 and s_2 are in the same 2-block.

There are at least $d_2(a_1) - 1$ a_1-components for s_1, $d_2(a_2) - 1$ a_2-components for s_1, $d_2(a_3)-1$ a_3-components for s_2, $d_2(a_4)-1$ a_4-components for s_2, $d_3(s_1)-1$ s_1-components that do not contain any s_2-component and $d_3(s_2)-1$ s_2-components that do not contain any s_1-component. Those components share only vertices in $\mathcal{A}$. Thus

$$w(G) \geq 2\sum_{i=1}^{4} d_2(a_i) - 8 + \sum_{i=1}^{2} d_3(s_i) - 2.$$

Case 4: $|\mathcal{A}| = 4$, s_1 and s_2 are not in the same 2-block and s_2 is in a 2-block that contains a_1 or a_2.

There are at least $d_2(a_1) - 1$ a_1-components for s_1, $d_2(a_2) - 2$ a_2-components for s_1 that do not contain any vertex in $\{a_3, a_4\}$, $d_2(a_3) - 1$ a_3-components for s_2, $d_2(a_4) - 1$ a_4-components for s_2, $d_3(s_1)$ s_1-components that do not contain any s_2-component and $d_3(s_2)$ s_2-components that do not contain any s_1-component. Those components share only vertices in $\mathcal{A}$. Thus

$$w(G) \geq 2\sum_{i=1}^{4} d_2(a_i) - 10 + \sum_{i=1}^{2} d_3(s_i).$$

Case 5: $|\mathcal{A}| = 4$, s_1 and s_2 are not in the same 2-block, and s_2 is in a 2-block that does not contain a_1 or a_2.

Without loss of generality, if s_2 is in any a_i-component, $i \in \{1,2\}$, for s_1, then let s_2 be in a_1-component for s_1. Similarly, if s_1 is in any a_i-component, $i \in \{3,4\}$, for s_2, then let s_2 be in a_3-component for s_1. There

are at least $d_2(a_1) - 2$ a_1-components for s_1 that do not contain s_2, $d_2(a_2) - 1$ a_2-components for s_1, $d_2(a_3) - 2$ a_3-components for s_2 that do not contain s_1, $d_2(a_4) - 1$ a_4-components for s_2, $d_3(s_1) - 1$ s_1-components that do not contain any s_2-component and $d_3(s_2) - 1$ s_2-components that do not contain any s_1-component Furthermore, if s_2 is in any a_i-component for s_1, $i \in \{1, 2\}$, we have another s_1-component that does not contain any s_2-component Otherwise, we have another a_1-component that does not contain s_2. Similar arguments can be applied on s_1. Thus

$$w(G) \geq 2\sum_{i=1}^{4} d_2(a_i) - 12 + \sum_{i=1}^{2} d_3(s_i) + x.$$

where $x = 0$ if (a_1, a_3) is a cut-edge; otherwise, $x \geq 1$.

We know that

$$\sum_{i=1}^{2} sd(s_i) \geq w(G). \tag{A.2}$$

If we substitute the lower bound value of $w(G)$ and the value of $\sum_{i=1}^{2} sd(s_i)$ (Equation A.1) into Inequality A.2, we can derive a contradiction except in case 5 when $x = 0$.

In case 5 when $x = 0$, we can derive $\sum_{i=1}^{4} d_2(a_i) = 6$. Since $|\mathcal{A}| = 4$ and two of the vertices in $\mathcal{A}$ are cutpoints, the other two vertices in $\mathcal{A}$ are not cutpoints. Furthermore, if G contains cutpoints other than those in $\mathcal{A}$, then

$$w(G) \geq 2\sum_{i=1}^{4} d_2(a_i) - 11 + \sum_{i=1}^{2} d_3(s_i) + x.$$

It is also true that if G contains any Tutte pair that does not include one of two cutpoints in $\mathcal{A}$, then

$$w(G) \geq 2\sum_{i=1}^{4} d_2(a_i) - 11 + \sum_{i=1}^{2} d_3(s_i) + x.$$

Thus the lemma holds. □

Proof of Lemma 5.4.3:

We prove the lemma by contradiction. Let $\Im_1 = \{c\}$ and let s_2 and s_3 be two distinct Tutte pairs in $\Im_2$. Let s_1 be a Tutte pair contains c. Let $s_1 = (a_1, a_2)$, $s_2 = (a_3, a_4)$ and $s_3 = (a_5, a_6)$.

Let $a_1 = c$ and let $\mathcal{A} = \{a_i \mid 1 \leq i \leq 6\}$. Note that none of the vertices in $\{a_i \mid 3 \leq i \leq 6\}$ could be cutpoints, since otherwise we can use the proof of Lemma 5.4.2 to get a contradiction. Note that

$$\sum_{i=1}^{3} sd(s_i) = \sum_{i=1}^{3} d_3(s_i) + \sum_{i=1}^{2} d_2(a_i) - 5.$$

We find out the possible values for $w(G)$ in the following cases.

Case 1: s_1, s_2 and s_3 are in the same 2-block.

There are at least a total of $\sum_{i=1}^{3} d_3(s_i) - 4$ s_1-components, s_2-components and s_3-components such that none of them is contained in the other. There are also $d_2(a_i) - 1$ a_i-components, $i \in \{1, 2\}$. Thus

$$w(G) \geq 2(\sum_{i=1}^{2} d_2(a_i) - 2) + \sum_{i=1}^{3} d_3(s_i) - 4.$$

Case 2: s_1 and s_2 are in a 2-block, while s_3 is in another 2-block.

If s_3 is in a 2-block that contains one of the vertices in $\{a_i \mid 1 \leq i \leq 4\}$, then there are at least $d_3(s_1) - 1$ s_1-components, $d_3(s_2) - 1$ s_2-components and $s_3(s_3) - 1$ s_3-components such that they share only vertices in $\mathcal{A}$. There are also at least $d_2(a_i) - 1$ a_i-components, $i \in \{1, 2\}$, with possibly one of the a_1-component contains s_3. Thus

$$w(G) \geq 2(\sum_{i=1}^{2} d_2(a_i) - 3) + \sum_{i=1}^{3} d_3(s_i) - 3.$$

Otherwise, from an argument similar to the above we can derive

$$w(G) \geq 2(\sum_{i=1}^{2} d_2(a_i) - 2) + \sum_{i=1}^{3} d_3(s_i) - 2.$$

Case 3: s_2 and s_3 are in a 2-block, while s_1 is in another 2-block.

If s_2 and s_3 are in an a_1-component or a_2-component, then there are $d_2(a_i)-1$ a_i-components, $i \in \{1,2\}$, with possibly one of them contains s_2 and s_3. There are also at least $d_3(s_i)-1$ s_i-components such that they share only vertices in $\mathcal{A}$. Thus

$$w(G) \geq 2(\sum_{i=1}^{2} d_2(a_i) - 3) + \sum_{i=1}^{3} d_3(s_i) - 3.$$

Otherwise,

$$w(G) \geq 2(\sum_{i=1}^{2} d_2(a_i) - 2) + \sum_{i=1}^{3} d_3(s_i) - 2.$$

Case 4: s_1, s_2 and s_3 are in distinct 2-blocks.

There are at least $d_3(s_i)-1$ s_i-components, $i \in \{2,3\}$, and $d_3(s_1)$ s_1-components such that they share only vertices in $\mathcal{A}$. There are at least $\sum_{i=1}^{2} d_2(a_i) - 4$ a_i-components, $i \in \{1,2\}$. Thus

$$w(G) \geq 2(\sum_{i=1}^{2} d_2(a_i) - 4) + \sum_{i=1}^{3} d_3(s_i) - 2.$$

We know that

$$\sum_{i=1}^{3} sd(s_i) \geq \frac{3}{2} w(G).$$

If we substitute the lower bound value of $w(G)$ and the value of $\sum_{i=1}^{3} sd(s_i)$ into the above inequality, we can derive a contradiction in each case except in case 4 when $d_2(a_1)=2$, $d_2(a_2)=1$ and $\sum_{i=2}^{3} d_3(s_i)=6$. But if this is the case, then $sd(s_1) > sd(s_2)$. Thus s_i could not be critical at the same time, $1 \leq i \leq 3$. This proves the lemma. The rest of the lemma can be proved in a similar way.

□

Proof of Lemma 5.4.4:

We prove the lemma by contradiction. Let $\Im_2 = \{s_1, s_2, s_3, \ldots, s_k\}$, where $k \geq 3$. Let $s_1 = (a_1, a_2)$, $s_2 = (a_3, a_4)$ and $s_3 = (a_5, a_6)$. Let $\mathcal{A} = \{a_i \mid 1 \leq i \leq 6\}$. Note that none of the Tutte pairs in $\{s_i \mid 1 \leq i \leq 3\}$ contains any cutpoint; otherwise, a contradiction can be derived using the proof of Lemma 5.4.3. Note that

$$\sum_{i=1}^{3} sd(s_i) = \sum_{i=1}^{3} d_3(s_i) - 3.$$

We find out the possible values for $w(G)$ in the following cases.

Case 1: s_1, s_2 and s_3 are in the same 2-block.

There are at least a total of $\sum_{i=1}^{3} d_3(s_i) - 4$ s_1-components, s_2-components and s_3-components such that none of them is contained in the other. Thus

$$w(G) \geq \sum_{i=1}^{3} d_3(s_i) - 4.$$

Case 2: s_1 and s_2 are in a 2-block, while s_3 is in another 2-block.

There are at least $d_3(s_i) - 1$ s_i-components, $1 \leq i \leq 3$, such that they only share vertices in $\mathcal{A}$. Thus

$$w(G) \geq \sum_{i=1}^{3} d_3(s_i) - 3.$$

Case 3: s_1, s_2 and s_3 are in distinct 2-blocks.

There are at least a total number of $\sum_{i=1}^{3} d_3(s_i) - 4$ s_i-components, $1 \leq i \leq 3$, with none of them contains more than one Tutte pair in $\{s_i \mid 1 \leq i \leq 3\}$. Thus

$$w(G) \geq \sum_{i=1}^{3} d_3(s_i) - 4.$$

We also know that

$$\sum_{i=1}^{3} sd(s_i) \geq \frac{3}{2} w(G).$$

If we substitute the lower bound value of $w(G)$ and the value of $\sum_{i=1}^{3} sd(s_i)$ into the above inequality, we can derive a contradiction in case 2. The resulting inequality holds for case 1 only when $d_3(s_i) = 2$, $1 \leq i \leq 3$ and $w(G) = 2$. Since the weight of G is greater than 2, the lemma holds. The resulting inequality holds for case 3 only when $d_3(s_i) = 2$, $1 \leq i \leq 3$. Since there are at least 3 2-blocks in 2-$blk(G)$, $w(G)$ is at least 4. Thus s_i, $1 \leq i \leq 3$, could not be critical. The rest of the lemma can be derived in a similar way. □

Proof of Lemma 5.4.14:

Let L be an c-component for s in G. Then 2-$blk(L)$ is a connected component in 2-$blk(G) - \{c\}$ which contains a degree-1 b-vertex r in 2-$blk(G)$. The corresponding 2-block H of r is not in any other c-component or s-component.

If L is a s-component in G, then let W be the 2-block that contains s. Then 3-$blk(L \cap W)$ is a connected component in the resulting forest obtained from 3-$blk(G)$ by removing the corresponding σ-vertex for s. Thus there is a β-vertex in 3-$blk(L \cap W)$ with degree 1 in 3-$blk(G)$. It is either the degree-1 β-vertex contains no cutpoint or it contains a cutpoint other than a_1 and a_2. If the degree-1 β-vertex contains no cutpoint, then we have proved the claims. If it contains a cutpoint c that is not equal to either a_1 or a_2, then we can find a b-vertex in 2-$blk(L \cup W)$ with a degree 1 in 2-$blk(G)$ using a similar argument.

□

Appendix B

Performance Data for Parallel Programs

program	n	m	test 1 (secs)	test 2 (secs)	test 3 (secs)	test 4 (secs)	average (secs)
Finding a spanning forest	256	16,382	0.71	0.69	0.67	0.69	0.69
	362	32,766	0.61	0.60	0.60	0.60	0.60
	512	65,534	1.35	1.81	1.35	1.82	1.58
	724	131,070	2.21	2.20	2.21	2.87	2.37
	1,024	262,142	4.61	6.06	4.65	6.06	5.35
	1,448	524,286	8.43	8.41	8.27	8.41	8.38
	2,048	1,048,574	17.17	17.16	17.18	22.61	18.53
	645	16,382	1.00	0.75	0.79	0.80	0.83
	1,024	32,766	0.88	0.83	0.86	1.06	0.91
	1,625	65,534	1.95	2.04	1.53	1.95	1.87
	2,580	131,070	2.98	3.81	2.99	3.78	3.39
	4,096	262,142	7.37	7.13	7.21	7.19	7.23
	6,501	524,286	14.44	14.41	14.40	14.32	14.39
	10,321	1,048,574	28.62	28.85	28.92	28.96	28.84
	10,922	16,382	3.02	4.30	4.09	3.00	3.60
	21,844	32,766	7.42	7.49	7.43	7.32	7.41
	43,688	65,534	14.53	19.14	14.27	14.65	15.65
	87,376	131,070	37.85	28.32	38.37	28.94	33.37
	131,070	196,605	59.93	44.95	44.68	46.82	49.09
	174,752	262,142	76.20	57.64	57.34	108.27	74.86
	349,504	524,286	153.00	154.13	154.23	115.77	144.28
	524,286	786,429	240.86	245.12	241.43	241.58	242.25

program	n	m	test 1 (secs)	test 2 (secs)	test 3 (secs)	test 4 (secs)	average (secs)
Finding	256	16,382	1.41	1.78	1.48	1.50	1.54
a minimum	362	32,766	1.60	1.58	2.02	2.10	1.83
spanning	512	65,534	4.47	4.71	5.54	4.38	4.78
forest	724	131,070	8.87	9.96	7.83	9.21	8.96
	1,024	262,142	18.48	20.41	18.72	21.17	19.69
	1,448	524,286	33.87	34.68	40.86	40.42	37.46
	2,048	1,048,574	71.28	69.70	67.29	63.19	67.86
	645	16,382	1.64	1.72	1.74	1.68	1.69
	1,024	32,766	2.33	2.21	2.46	2.42	2.35
	1,625	65,534	4.61	5.49	4.14	4.70	4.74
	2,580	131,070	9.98	7.53	9.40	9.28	9.05
	4,096	262,142	18.60	19.15	15.15	21.43	18.58
	6,501	524,286	39.72	37.34	44.51	39.39	40.24
	10,321	1,048,574	91.13	83.77	80.02	78.21	83.28
	10,922	16,382	4.25	3.87	3.33	4.40	3.96
	21,844	32,766	5.53	7.46	5.04	7.23	6.32
	43,688	65,534	14.23	11.77	17.92	14.12	14.51
	87,376	131,070	24.62	26.02	30.01	24.88	26.38
	174,752	262,142	56.67	45.24	44.91	61.05	51.97
	349,504	524,286	81.99	112.32	112.33	101.43	102.02
	699,048	1,048,572	220.88	212.18	225.07	244.75	225.72

program	n	m	test 1 (secs)	test 2 (secs)	test 3 (secs)	test 4 (secs)	average (secs)
Finding an ear decomposition	256	16,382	1.17	1.12	1.12	1.14	1.14
	362	32,766	1.11	1.13	1.13	1.13	1.13
	512	65,534	2.84	2.36	2.84	2.39	2.61
	724	131,070	4.19	4.21	4.88	4.17	4.36
	1,024	262,142	8.78	8.66	8.77	8.64	8.71
	1,448	524,286	17.91	17.86	17.83	17.85	17.86
	2,048	1,048,574	41.33	45.99	46.04	41.24	43.65
	645	16,382	1.51	1.25	1.27	1.24	1.32
	1,024	32,766	1.39	1.41	1.61	1.61	1.51
	1,625	65,534	3.08	2.62	2.65	2.59	2.74
	2,580	131,070	6.01	5.22	5.18	5.98	5.60
	4,096	262,142	11.58	11.65	11.74	10.32	11.32
	6,501	524,286	22.82	23.15	23.08	23.01	23.02
	10,321	1,048,574	46.23	46.40	40.69	46.57	44.97
	21,844	32,766	8.88	8.87	8.37	8.79	8.73
	43,688	65,534	17.79	17.33	17.58	17.31	17.50
	87,376	131,070	34.28	33.74	33.60	43.58	36.30
	131,070	196,605	52.57	52.51	68.60	53.58	56.82
	174,752	262,142	87.51	67.33	67.76	67.56	72.54
	349,504	524,286	138.33	177.21	176.93	176.67	167.28
	524,200	786,300	268.98	203.11	269.47	266.96	252.13
Finding an open ear decomposition	256	16,382	3.46	3.28	3.41	3.35	3.37
	362	32,766	3.86	3.97	4.12	4.11	4.02
	512	65,534	9.45	8.77	8.82	8.93	8.99
	724	131,070	16.14	15.12	15.29	15.94	15.62
	1,024	262,142	33.13	33.54	34.79	32.55	33.50
	1,448	524,286	70.69	70.67	69.53	70.77	70.42
	645	16,382	3.72	3.22	3.38	3.52	3.46
	1,024	32,766	4.16	4.21	4.19	4.31	4.22
	1,625	65,534	9.42	7.81	7.99	7.65	8.22
	2,580	131,070	17.22	18.14	17.09	16.06	17.13
	4,096	262,142	31.40	33.72	35.29	34.35	33.69
	6,501	524,286	65.62	64.89	70.74	66.14	66.85
	10,922	16,382	6.21	6.11	4.75	5.92	5.75
	21,844	32,766	10.65	10.56	10.13	10.50	10.46
	43,688	65,534	20.97	21.46	20.88	21.11	21.10
	87,376	131,070	52.03	42.99	42.45	42.22	44.92
	174,752	262,142	105.81	85.11	85.43	85.03	90.35
	262,100	393,150	115.50	118.20	116.57	118.14	117.10

program	n	m	test 1 (secs)	test 2 (secs)	test 3 (secs)	test 4 (secs)	average (secs)
Finding all cut edges	256	16,382	1.93	1.87	1.88	1.89	1.89
	362	32,766	1.87	1.87	1.86	1.88	1.87
	512	65,534	4.34	3.88	4.35	3.88	4.11
	724	131,070	7.19	7.21	7.87	7.20	7.37
	1,024	262,142	16.61	15.12	16.57	15.10	15.85
	1,448	524,286	29.73	29.67	29.55	29.70	29.66
	2,048	1,048,574	65.96	63.89	63.94	69.38	65.79
	645	16,382	2.01	1.98	1.99	2.15	2.03
	1,024	32,766	2.04	2.05	2.03	2.09	2.05
	1,625	65,534	4.47	4.46	4.45	3.98	4.34
	2,580	131,070	8.78	8.00	7.96	8.80	8.39
	4,096	262,142	17.87	17.84	17.94	18.04	17.92
	6,501	524,286	34.95	32.01	35.00	31.97	33.48
	10,321	1,048,574	71.94	66.04	71.99	71.60	70.39
	10,922	16,382	5.45	5.41	5.19	5.32	5.34
	21,844	32,766	8.93	9.27	11.19	8.95	9.59
	43,688	65,534	17.86	17.02	17.91	17.63	17.60
	87,376	131,070	35.78	35.57	35.49	34.21	35.26
	131,070	196,605	71.62	70.59	71.60	70.64	71.11
	174,752	262,142	88.49	88.28	68.95	87.72	83.36
	349,504	524,286	137.68	177.80	178.97	178.35	168.20
	524,286	786,429	277.93	278.59	282.71	281.90	280.28
Finding a strong orientation	256	16,382	1.18	1.17	1.17	1.18	1.18
	362	32,766	1.15	1.16	1.16	1.16	1.16
	512	65,534	2.93	2.45	2.93	2.48	2.70
	724	131,070	4.28	4.31	4.97	4.26	4.45
	1,024	262,142	9.00	9.12	9.00	9.12	9.06
	1,448	524,286	18.10	18.08	18.04	18.07	18.08
	2,048	1,048,574	41.24	45.89	45.95	41.15	43.56
	645	16,382	1.53	1.28	1.30	1.26	1.34
	1,024	32,766	1.42	1.44	1.63	1.64	1.53
	1,625	65,534	3.13	2.67	2.70	2.64	2.78
	2,580	131,070	6.10	5.31	5.27	6.06	5.68
	4,096	262,142	10.58	11.87	11.94	12.03	11.61
	6,501	524,286	23.32	23.67	23.60	23.52	23.53
	10,321	1,048,574	47.25	47.38	41.60	47.55	45.94
	21,844	32,766	8.98	8.97	8.47	8.89	8.82
	43,688	65,534	17.96	17.51	17.76	17.49	17.68
	87,376	131,070	36.14	35.57	35.42	46.03	38.29
	131,070	196,605	52.84	52.75	68.86	53.82	57.06
	174,752	262,142	90.91	69.93	70.38	70.17	75.35
	349,504	524,286	143.04	183.32	183.06	182.79	173.05
	524,200	786,300	263.55	262.95	199.05	261.08	246.66

BIBLIOGRAPHY

[AM86] R. J. Anderson and G. L. Miller. Optimal parallel algorithms for list ranking. Extended abstract, 1986.

[AM88] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. In *Proc. 3rd Aegean Workshop on Computing*, volume LNCS #319, pages 81–90. Springer-Verlag, 1988.

[Ama83] A. Amar. On the connectivity of some telecommunications networks. *IEEE Trans. on Computers*, C-32(5):512–519, 1983.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Spring Joint Computer Conf.*, pages 483–485, 1967.

[AS87] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Tran. on Computers*, pages 1258–1263, October 1987.

[AS92] R. Anderson and J. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 168–177, 1992.

[ASE92] N. Alon, J. H. Spencer, and P. Erdös. *The Probabilistic Method.* John Wiley & Sons, Inc., 1992.

[BBM90] D. Bienstock, E. F. Brickell, and C. L. Monma. On the structure of minimum-weight k-connected spanning networks. *SIAM J. Disc. Math.*, 3(3):320–329, 1990.

[Ble89] G. E. Blelloch. *Scan Primitives and Parallel Vector Models*. PhD thesis, M.I.T., October 1989.

[BLM+91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. 3th ACM Symp. on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[BR84] S. H. Bokhari and A. D. Raza. Augmenting computer networks. In *Proc. Int'l Conf. on Parallel Processing*, pages 338–345, 1984.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, 1974.

[BST77] F. T. Boesch, C. Suffel, and R. Tindell. The spanning subgraphs of Eulerian graphs. *J. Graph Theory*, 1:79–84, 1977.

[CBKT93] R. F. Cohen, G. Di Battista, A. Kanevsky, and R. Tamassia. Reinventing the wheel: an optimal data structure for connectivity queries. In *Proc. 25th Annual ACM Symp. on Theory of Comp.*, pages 194–200, 1993.

[Che52] H. Chernoff. A measure of the asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of mathematical Statistics*, 23:493–509, 1952.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th Symp. on Principles and Practices of Parallel Programming*, pages 1–12, 1993.

[CL93] K. W. Chong and T. W. Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. In *Proc. 4th Annual ACM-SIAm Symp. on Discrete Algorithms*, pages 11–20, 1993.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[Col88] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17:770–785, 1988.

[CS89] G.-R. Cai and Y.-G. Sun. The minimum augmentation of any graph to a k-edge-connected graph. *Networks*, 19:151–172, 1989.

[CV86] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proc. 27th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 478–491, 1986.

[CV88] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, 17:128–142, 1988.

[CW81] N. Christofides and C. A. Whitlock. Network synthesis with connectivity constraints — a survey. In *Operational Research '81*, pages 705–723, 1981.

[DBT90] G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In *Proc. 17th Int'l Conf. on Automata, Language and Programming*, volume LNCS # 443, pages 598–611. Springer-Verlag, 1990.

[DKL76] E. A. Dinits, A. V. Karzanov, and M. L. Lomosonov. On the structure of a family of minimal weighted cuts in a graph. In A. A. Fridman, editor, *Studies in Discrete Optimization*, pages 290–306. Nauka, Moscow, 1976. in Russian.

[DL92a] B. Dixon and A. K. Lenstra. Factoring integers using SIMD sieves. Manuscript, 1992.

[DL92b] B. Dixon and A. K. Lenstra. Massively parallel elliptic curve factoring. Manuscript, 1992.

[DMM92] N. Dean, M. Mevenkamp, and C. L. Monma. NETPAD: An interface graphics system for network modeling and optimization. In *Proc. Computer Science & Operations Research: New Developments in their Interfaces*, pages 231–243. Pergamon Press, 1992.

[DNS81] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Comput.*, 10:657–675, 1981.

[Eck79] D. M. Eckstein. Simultaneous memory access. Technical report, Computer Science Dept., Iowa State Univ., Ames, IA, 1979.

[Esw73] K. P. Eswaran. Representation of graphs and minimally augmented Eulerian graphs with applications in data base management. Technical Report RJ 1305, IBM, Yorktown Heights, N.Y., 1973.

[ET75] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[ET76] K. P. Eswaran and R. E. Tarjan. Augmentation problems. *SIAM J. Comput.*, 5(4):653–665, 1976.

[Eve79] S. Even. *Graph Algorithms.* Computer Science Press, Rockville, MD, 1979.

[FBW89] D. Fernández-Baca and M. A. Williams. Augmentation problems on hierarchically defined graphs. In *1989 Workshop on Algorithms and Data Structures*, volume LNCS # 382, pages 563–576. Springer-Verlag, 1989.

[FC70] H. Frank and W. Chou. Connectivity considerations in the design of survivable networks. *IEEE Trans. on Circuit Theory*, CT-17(4):486–490, December 1970.

[FJ81] G. N. Frederickson and J. JáJá. Approximation algorithms for several graph augmentation problems. *SIAM J. Comput.*, 10(2):270–283, May 1981.

[FJ93] A. Frank and T. Jordán. Minimal edge-coverings of pairs of sets. Manuscript, June 1993.

[Fra90] A. Frank. Augmenting graphs to meet edge-connectivity requirements. In *Proc. 31th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 708–718, 1990.

[Fra92] A. Frank. Augmenting graphs to meet edge-connectivity requirements. *SIAM J. Disc. Math.*, 5(1):25–43, February 1992.

[FRT93] D. Fussel, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacements. *SIAM J. Comput.*, 22(3):587–616, 1993.

[FS71] D. R. Fulkerson and L. S. Sharpley. Minimal k-arc connected graphs. *Networks*, 1:91–98, 1971.

[Gab91] H. N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proc. 32th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 812–821, 1991.

[GGW93] H. N. Gabow, M. X. Goemans, and D. P. Williamson. An efficient approximation algorithm for the survivable network design problem. In *Proc. 3rd IPCO Conference*, 1993, to appear.

[GH73] S. Goodman and S. Hedetniemi. On the Hamiltonian completion problem, 1973. Presented at the Capital Conf. on Graph Theory and Combinatorics, Washington, D. C.

[GH85] R. L. Graham and P. Hell. On the history of the minimum spanning problem. *Annels of the History of Computing*, 7(1):43–57, 1985.

[GI91] Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *ACM SIGACT News*, 22(1):57–61, 1991.

[GJ79] M. R. Garey and D. S. Johnson. *COMPUTERS AND INTRACTABILITY A Guide to the Theory of* **NP**-*Completeness*. W. H. Freeman and Company, New York, 1979.

[GM90] M. Grötschel and C. L. Monma. Integer polyhedra associated with certain network design problems with connectivity constraints. *SIAM J. Disc. Math.*, 3:502–523, 1990.

[GMR93] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW pram: Accounting for contention in parallel algorithms. Manuscript, July, 1993.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge Univ. Press, 1988.

[Gus87] D. Gusfield. Optimal mixed graph augmentation. *SIAM J. Comput.*, 16(4):599–612, August 1987.

[Gus89] D. Gusfield. A graph theoretic approach to satistical data security. *SIAM J. Comput.*, 75:552–571, 1989.

[GW92] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. In *Proc. 3rd Annual ACM-SIAM Symp. on Discrete Alg.*, pages 307–315, 1992.

[Hag87] T. Hagerup. Towards optimal parallel bucket sorting. *Information and Computation*, 75:39–51, 1987.

[Har62] F. Harary. The maximum connectivity of a graph. *Proc. Nat. Acad. Sci.*, 48:1142–1146, 1962.

[Har69] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.

[HKRT92] X. Han, P. Kelsen, V. Ramachandran, and R. E. Tarjan. Computing minimal spanning subgraphs in linear time. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 146–156, 1992.

[HPR92] W. Hightower, J. Prins, and J. Reif. Implementations of randomized sorting on large parallel machines. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 158–167, 1992.

[HR91a] T.-s. Hsu and V. Ramachandran. A linear time algorithm for triconnectivity augmentation. In *Proc. 32th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 548–559, 1991.

[HR91b] T.-s. Hsu and V. Ramachandran. On finding a smallest augmentation to biconnect a graph. In *Proc. 2nd Annual Int'l Symp. on Algorithms*, volume LNCS #557, pages 326–335. Springer-Verlag, 1991. *SIAM J. Comput.*, to appear.

[HRD92] T.-s. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on the MasPar. In *AMS Proc. of DIMACS Workshop on Computational Support for Discrete Math.*, to appear. Also available as TR-92-38, Dept. of Comp. Sci., Univ. of Texas at Austin.

[HRD93] T.-s. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing. Technical Report TR-93-14, Dept. of Computer Sciences, Univ. of Texas at Austin, 1993.

[HS86] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29:1170–1183, 1986.

[Hsu92] T.-s. Hsu. On four-connecting a triconnected graph. In *Proc. 33th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 70–79, October 1992.

[HT73] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2:135–158, 1973.

[IEE91] IEEE Computer. New products column, January 1991.

[ISO85] M. Imase, T. Soneoka, and K. Okada. Connectivity of regular directed graphs with small dimeters. *IEEE Trans. on Computers*, C-34(3):267–273, 1985.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[JG86] S. P. Jain and K. Gopal. On network augmentation. *IEEE Trans. on Reliability*, R-35(5):541–543, 1986.

[Jor92] T. Jordán, February 1992. private communications.

[Jor93a] T. Jordán. Increasing the vertex-connectivity in directed graphs. In *Proc. 1st Euorpean Symp. on Algorithms*, 1993, to appear.

[Jor93b] T. Jordán. Optimal and almost optimal algorithms for connectivity augmentation problems. In *Proc. 3rd IPCO Conference*, 1993, to appear.

[Kan88] A. Kanevsky. *Vertex Connectivity of Graphs: Algorithms and Bounds.* PhD thesis, University of Illinois, Urbana-Champaign, IL, 1988. (Tech. Rep. ACT-97-UILU-ENG-88-2247, Coordinated Science lab.).

[Kan91] G. Kant. Linear planar augmentation algorithms for outerplanar graphs. Tech. Rep. RUU-CS-91-47, Dept. of Computer Science, Utrecht University, the Netherlands, 1991.

[Kan93] G. Kant. *Algorithms for Drawing Planar Graphs.* PhD thesis, Utrecht University, the Netherlands, 1993.

[KB91] G. Kant and H. L. Bodlaender. Planar graph augmentation problems. In *Proc. 2nd Workshop on Data Structures and Algorithms*, volume LNCS #519, pages 286–298. Springer-Verlag, 1991.

[KB92] G. Kant and H. L. Bodlaender. Triangulating planar graphs while minimizing the maximum degree. In *Proc. 3rd Scand. Workshop on Algorithm Theory*, volume LNCS #621, pages 258–271. Springer-Verlag, 1992.

[Kel92] P. Kelsen. *Efficient Computation of Extremal Structures in Graphs and Hypergraphs.* PhD thesis, University of Illinois, Urbana-Champaign, IL, 1992.

[KF79] T. Kashiwabara and T. Fujisawa. NP-completeness of the problem of finding a minimum-clique-number interval graph containing a given graph as a subgraph. In *Proc. of 1979 IEEE Int'l Symp. on Circuits and Systems*, pages 657–660, 1979.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming language.* Prentice Hall, Englewood Cliffs, NJ, 1978.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming language.* Prentice Hall, Englewood Cliffs, NJ, 1988. Second Edition.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 869–941. North Holland, 1990.

[KR91a] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. *J. Comp. System Sci.*, 42:288–306, 1991.

[KR91b] P. Kelsen and V. Ramachandran. On finding minimal 2-connected subgraphs. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 178–187, 1991.

[KT86] A. V. Kazanov and E. A. Timofeev. Efficient algorithm for finding all minimal edge cuts of a nonoriented graph. *Cybernetics*, pages 156–162, 1986. Translated from Kibernetika, No. 2, pp. 8–12, March–April 1986.

[KT91] A. Kanevsky and R. Tamassia, October 1991. private communications.

[KT92] S. Khuller and R. Thurimella. Approximation algorithms for graph augmentation. In *Proc. 19th Int'l Conf. on Automata, Language and Programming*, volume LNCS #623, pages 330–341. Springer-Verlag, 1992.

[KTDBC91] A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen. On-line maintenance of the four-connected components of a graph. In *Proc. 32th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 793–801, 1991.

[KU86] Y. Kajitani and S. Ueno. The minimum augmentation of a directed tree to a k-edge-connected directed graph. *Networks*, 16:181–197, 1986.

[KV92] S. Khuller and U. Vishkin. Biconnectivity approximations and graph carvings. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 759–770, 1992.

[LAD+92] C. Leiserson, Z. S. Abuhamdeh, D. Douglas, C. R. Feynmann, M. Ganmukhi, J. Hill, W. D. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S-W Yang, and R. Zak. The network architecture of the Connection Machine CM-5. In *Proc. 4th ACM*

Symp. on Parallel Algorithms and Architectures, pages 272–287, 1992.

[Lam86] L. Lamport. LaTeX *A Document Preparation System.* Addison-Wesley, Reading, MA, 1986.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[LF80] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27:831–838, 1980.

[Mas91a] MasPar Computer Co. *MasPar Parallel Application Language (MPL) Reference manual*, version 2.0 edition, March 1991.

[Mas91b] MasPar Computer Co. *MasPar Parallel Application Language (MPL) User Guide*, version 2.0 edition, March 1991.

[Mas91c] MasPar Computer Co. *MasPar System Overview*, version 2.0 edition, March 1991.

[Mas92a] MasPar Computer Co. *MasPar Data Display Library (MPDDL) Reference manual*, version 3.0, rev. a6 edition, July 1992.

[Mas92b] MasPar Computer Co. *MasPar Parallel Application Language (MPL) Reference manual*, version 3.0, rev. a3 edition, July 1992.

[Mas92c] MasPar Computer Co. *MasPar Parallel Application Language (MPL) User Guide*, version 3.1, rev. a3 edition, November 1992.

[Mas92d] MasPar Computer Co. *Release Notes for MasPar System Software*, version 3.1, rev. b4 edition, November 1992.

[Mat72] D. W. Matula. k-components, clusters, and slicings in graphs. *SIAM J. Appl. Math.*, 22(3):459–480, 1972.

[Mat76] D. W. Matula. Subgraph connectivity numbers of a graph. In Y. Alavi and D. R. Lick, editors, *Theory and Applications of Graphs*. Springer-Verlag Lecture Notes in Mathematics, 1976.

[Mat78] D. W. Matula. k-blocks and ultrablocks in graphs. *J. Combinatorial Theory*, Series B, 24:1–13, 1978.

[MDM91] M. Mevenkamp, N. Dean, and C. L. Monma. *NETPAD User's Guide*. Bellcore, August 1991.

[Men27] K. Menger. Zur allgemeinen kurventheorie. *Fund. Math.*, 10:96–115, 1927.

[Mev91a] M. Mevenkamp. *NETPAD Programmer's Guide*. Bellcore, August 1991.

[Mev91b] M. Mevenkamp. *NETPAD Reference Guide*. Bellcore, August 1991.

[MHT87] T. Masuzawa, K. Hagihara, and N. Tokura. An optimal time algorithm for the k-vertex-connectivity unweighted augmentation problem for rooted directed trees. *Discrete Applied Mathematics*, pages 67–105, 1987.

[MMP90] C. L. Monma, B. S. Munson, and W. R. Pulleyblank. Minimum-weight two-connected spanning networks. *Math. Programming*, 46:153–171, 1990.

[MN89] K. Mehlhorn and S. Näher. LEDA, a library of efficient data types and algorithms. Technical Report TR A 04/89, FB10, Universität des Saarlaners, Saarbrücken, 1989.

[Mo88] Z. Mo. *Graph and Directed Graph Augmentation Problems*. PhD thesis, Western Michigan University, 1988.

[MR82] G. Memmi and Y. Raillard. Some new results about the (d,k) graphs problem. *IEEE Trans. Computers*, C-31:784–791, 1982.

[MR92] G. Miller and V. Ramachandran. A new triconnectivity algorithm and its applications. *Combinatorica*, 12:53–76, 1992.

[MR93] P. D. MacKenzie and V. Ramachandran. Optical communication and ERCW PRAMs. Manuscript, 1993.

[MSV86] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoret. Comput. Sci.*, pages 277–298, 1986.

[NGM90] D. Naor, D. Gusfield, and C. Martel. A fast algorithm for optimally increasing the edge-connectivity. In *Proc. 31th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 698–707, 1990.

[NM82] D. Nath and N. Maheshwari. Parallel algorithms for the connected components and minimal spanning tree problems. *Information Processing Letters*, 14(1):7–11, March 1982.

[NT92] B. Narendran and P. Tiwari. Polynomial root-finding: Analysis and computational investigation of a parallel algorithm. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 178–187, 1992.

[Nye88] A. Nye. *X Window System User's Guide.* O'Reilly & Associates, Inc., 1988.

[OMKF81] T. Ohtsuki, H. Mori, T. Kashiwabara, and T. Fujisawa. On minimal augmentation of a graph to obtain an interval graph. *J. Comp. System Sci.*, 22:60–97, 1981.

[PC93] R. Pickering and J. Cook. A first course in programming the DECmpp/Sx. Technical report, para//lab, Dept. of Informatics, Univ. of Bergen, N-5020 Bergen, Norway, 1993. Series of Parallel Processing: A Self-Study Introduction.

[Ple76] J. Plesník. Minimum block containing a given graph. *ARCHIV DER MATHEMATIK*, XXVII:668–672, 1976.

[Pre93a] L. Prechelt. Comparison of MasPar MP-1 and MP-2 communication operations. Technical Report 16/93, Institute für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, Germany, April 1993.

[Pre93b] L. Prechelt. Measurements of MasPar MP-1216A communication operations. Technical Report 01/93, Institute für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, Germany, January 1993.

[PS83] C. H. Papadimitriou and K. Steiglitz. *Combinational Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, NJ, 1983.

[PS90] J. F. Prins and J. A. Smith. Parallel sorting of large arrays on the MasPar MP-1. In *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, pages 59–64, 1990.

[Qui87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, 1987.

[Ram90] V. Ramachandran. Class notes. Dept. of Computer Sciences, Univ. of Texas at Austin, Spring 1990.

[Ram93] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 275–340. Morgan-Kaufmann, 1993.

[Rei93] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan-Kaufmann, 1993.

[RG77] A. Rosenthal and A. Goldner. Smallest augmentations to biconnect a graph. *SIAM J. Comput.*, 6(1):55–66, March 1977.

[RK93] R. Ravi and P. Klein. When cycles collapse: A general approximation technique for constrained two-connectivity problems. In *Proc. 1st Euorpean Symp. on Algorithms*, 1993, to appear.

[RR89] V. Ramachandran and J. Reif. An optimal parallel algorithm for graph planarity. In *Proc. 30th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 282–287, 1989.

[RT74] D. M. Ritchie and K. Thompson. The Unix timesharing system. *Communications of the ACM*, 17:365–375, July 1974.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Trans. on Programming Languages and Systems*, 2:484–521, October 1980.

[Sch84] U. Schumacher. An algorithm for construction of a k-connected graph with minimum number of edges and quasiminimal diameter. *Networks*, 14:63–74, 1984.

[Sor88] D. Soroker. Fast parallel strong orientation of mixed graphs and related augmentation problems. *Journal of Algorithms*, 9:205–223, 1988.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. In *Proc. 3rd Aegean Workshop on Computing*, volume LNCS #319, pages 111–123. Springer-Verlag, 1988.

[SWK69] K. Steiglitz, P. Weiner, and D. J. Kleitman. The design of minimum-cost survivable networks. *IEEE Trans. on Circuit Theory*, CT-16(4):455–460, 1969.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM Press, Philadelphia, PA, 1983.

[Tut66] W. T. Tutte. *Connectivity in Graphs*. University of Toronto Press, 1966.

[TV85] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14:862–874, 1985.

[UKW88] S. Ueno, Y. Kajitani, and H. Wada. Minimum augmentation of a tree to a k-edge-connected graph. *Networks*, 18:19–25, 1988.

[Vis83] U. Vishkin. Implementation of simultaneous memory address access in models that forbid it. *J. Algorithms*, 4:45–50, 1983.

[Vis84] U. Vishkin. Randomized speed-ups in parallel computation. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 230–239, 1984.

[Vis91] U. Vishkin. Structural parallel algorithmics. In *Proc. 18th ICALP*, volume LNCS #510, pages 363–380. Springer-Verlag, 1991.

[Wat87] T. Watanabe. An efficient way for edge-connectivity augmentation. Tech. Rep. ACT-76-UILU-ENG-87-2221, Coordinated Science lab., University of Illinois, Urbana, IL, 1987.

[WHN90] T. Watanabe, Y. Higashi, and A. Nakamura. Graph augmentation problems for a specified set of vertices. In *Proc. 1st Annual Int'l Symp. on Algorithms*, volume LNCS #450, pages 378–387. Springer-Verlag, 1990. Earlier version in Proc. 1990 Int'l Symp. on Circuits and Systems, pages 2861–2864.

[WMT92] T. Watanabe, T. Mashima, and S. Taoka. The k-edge-connectivity augmentation problem of weighted graphs. In *Proc. 3rd Annual Int'l Symp. on Algorithms and Computation*, volume LNCS #650, pages 31–40. Springer-Verlag, 1992.

[WN87] T. Watanabe and A. Nakamura. Edge-connectivity augmentation problems. *J. Comp. System Sci.*, 35:96–144, 1987.

[WN88] T. Watanabe and A. Nakamura. 3-connectivity augmentation problems. In *Proc. of 1988 IEEE Int'l Symp. on Circuits and Systems*, pages 1847–1850, 1988.

[WN90] T. Watanabe and A. Nakamura. A smallest augmentation to 3-connect a graph. *Discrete Applied Mathematics*, 28:183–186, 1990.

[WN93] T. Watanabe and A. Nakamura. A minimum 3-connectivity augmentation of a graph. *J. Comp. System Sci.*, 46:91–128, 1993.

[WNN89] T. Watanabe, T. Narita, and A. Nakamura. 3-edge-connectivity augmentation problems. In *Proc. of 1989 IEEE Int'l Symp. on Circuits and Systems*, pages 335–338, 1989.

[Wol88] S. Wolfram. *Mathematica*™ *A System for Doing Mathematics by Computer*. Addison-Wesley, 1988.

[WYO91] T. Watanabe, M. Yamakado, and K. Onaga. A linear time augmenting algorithm for 3-edge-connectivity augmentation problems. In *Proc. of 1991 IEEE Int'l Symp. on Circuits and Systems*, pages 1168–1171, 1991.

VITA

Tsan-sheng Hsu was born in Sanchung, Taipei, Taiwan, the Republic of China, on July 29, 1963, the second son of Gan-hua Lu and Wu-toun Hsu. He studied at the National Taiwan University, from 1981 to 1985, where he received a Bachelor of Science degree in Computer Sciences. After graduation, he served the compulsory two-year military service as an army infantry officer. From August 1987 to July 1988, he was a teaching assistant and a computer system manager at the Department of Computer Science and Information Engineering, National Taiwan University. He then entered the Ph.D. program of the University of Texas at Austin in September 1988. He received a Master of Science in Computer Sciences degree in May 1990 and he expects to receive the Ph.D. degree in Computer Sciences in Fall 1993.

Permanent address: No. 7 Alley 15 Lane 422
Jen-Ai Street, Sanchung
Taipei 24149, Taiwan
Republic of China

This dissertation was typeset[1] with LaTeX by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin dissertation format was written by Khe-Sing The and revised by Tsan-sheng Hsu.