PARALLEL MATROID ALGORITHMS

by

Ada Fu

B.Sc., Chinese University of Hong Kong, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computing Science

SIMON FRASER UNIVERSITY

May 1985

ISBN Ø-315-3Ø727-7

APPROVAL

Name : Ada Wai-Chee Fu

Degree : Master of Science

Title of Thesis : Parallel Matroid Algorithms

Examining Committee :

Chairperson : James J. Weinkam

Senior Supervisor: Joseph Peters

Tiko Kameda

Arthur Lee Liestman

External Examiner: Richard Anstee
Department of Mathematics
University of British Columbia

Date Approved: 27 May 1985

2

ABSTRACT

Using the Fetch-and-Add CRCW PRAM model, parallel algorithms are designed in this thesis for general weighted and unweighted two-matroid intersection problems, and for two special two-matroid intersection problems: the bipartite matching problem (both weighted and unweighted), and the directed spanning tree problem. All of these parallel algorithms achieve perfect speed up with respect to the corresponding sequential algorithms. Fast algorithms but without perfect speed up are also designed.

Two new parallel write operations, "write-max" and "write-min", are introduced. Using these new operations, the time complexities of some of the algorithms mentioned above can be improved by an O(log n) factor where n is the number of elements in the matroids. Implementation of these operations is shown to be simpler than Fetch-and-Add. Also, a generalization of the CRCW and Fetch-and-Add CRCW PRAM models is suggested which gives rise to a more powerful model. We call this the "concurrent critical section model". It can be shown that simulation of this model by a CREW PRAM model has time and space complexities of O(Tlogn) and O(Mnlogn) respectively, where n is the number of processors, and T and M are the time and number of variables involved in the critical section. These are the same time and space complexities as the simulation of the Fetch-and-Add CRCW PRAM model by the CREW PRAM model. We believe that the write-max and the write-min operations and the concurrent critical section model will be very useful for designing parallel algorithms for other difficult problems.

To my parents.

ACKNOWLEDGEMENTS

I would like to express my deepest thanks to my supervisor, Professor Joseph Peters, for his patience, guidance and encouragement during the past year for preparation of this thesis, and for all the time and effort he put into reading the many drafts of the thesis, giving valuable advice and corrections for both the research results and English errors.

I would also like to thank members of my committee, Professors Art Liestman and Tiko Kameda for reading and correcting my thesis, and in particular Professor Richard Antsee for his careful reading and suggestions in some mathematical aspects of the thesis. I would like to thank the faculty and staff of the department who have contributed greatly in creating a environment favorable to carrying out my research work.

Grateful acknowledgement is given to Simon Fraser University and the Department of Computing Science for financial support.

CONTENTS

1. INTRODUCTION

1.1. Motivation

Conventional sequential computing systems are approaching intrinsic physical limitations on their computing speed. One way to achieve higher speeds in computing systems is parallelism. With the advent of monolithic VLSI technology, the production of low-cost single chip processors is possible and we can build parallel computers consisting of thousands of such processors. As Zakharov remarks in his paper [Za-84]:

"Looking into the future, it seems that we are in fact in a transition phase from purely sequential systems and that parallelism will become a standard feature of most computer systems in the future at the processor level."

Parallel processing is a very promising trend and we may expect the next generation of computers to be parallel computers which may bring about a revolution in practically every aspect of computer science.

However, no one knows how to build a parallel computer in the best possible way. There are many alternatives on how we can connect the processors and memories, and on what kinds of operations we allow for parallel processing. To make reasonable decisions on these questions, we have to look at the possible effects of these choices on the design of algorithms. In this thesis, a model which consists of multiple processors and shared memory is used because it is suitable for algorithms with complex data flows. The thesis aims at designing parallel algorithms for the two-matroid intersection problems using this parallel model.

Three objectives are achieved:

(1) The first objective is to understand more about the the power of parallelism in computer processing. Before we can justify the introduction of parallel computers, we must find out how far we can parallelize the solution of problems of interest.

There are two kinds of measures of the performance of parallel algorithms. The first is the measure of speed-up of the parallel algorithm over an efficient sequential algorithm. If T_p

is the parallel time and T_s is the sequential time, then the speed-up ratio, R_s, is $\frac{T_s}{T_p}$. If x processors are used in the parallel processing, then a *perfect speed-up* is achieved if x equals $O(R_s)$. Equivalently, we may define the *time-processor product* as T_p times x, and a perfect speed-up is achieved if the time-processor product is equal to $O(T_s)$.

The second measure is the parallel time itself. A parallel algorithm which runs in logarithmic time or sub-linear time using a polynomial number of processors is usually considered to be a very fast algorithm and will well justify the use of parallelism. It is generally believed that some problems in P cannot be solved in log-time in parallel, where P is the class of problems solvable in polynomial time sequentially. It has been proved in [Go-78] that if a problem is log-space complete for P, and if it can be solved in log-time in parallel, then all problems in P can be solved in log-time in parallel. Hence we can prove that a problem is unlikely to be solvable in logarithmic time by proving that it is "log-space complete" for P.

In order to challenge the power of parallelism, the two-matroid intersection problems are chosen. So far, parallel algorithms have been designed predominantly for simple problems or for problems with regular data structures. Two-matroid intersection problems are in one sense some of the hardest combinational problems that can be solved by polynomial time algorithms [PS-82]. The single-matroid problems have an efficient parallel algorithm which runs in log-time (chapter 3). The three-matroid intersection problems are NP-complete. The two-matroid intersection problems can be solved in polynomial time but the sequential algorithms for these problems have complex structures and apparent sequential natures. It is not known whether the two-matroid problems can be solved in sub-linear time.

The two-matroid problems can be viewed as linear programming problems. In fact, one of the fastest known sequential algorithms is based on linear programming techniques [La-76],[PS-82]. In [DLR-79], linear programming problems have been shown to be log-space hard for P. Also, the fastest sequential algorithm for the bipartite matching problem, which is a special case of the two-matroid problems, is based on a transformation into a max-flow problem [Ta-82]. The max-flow problem has been shown to be log-space complete for P in [GSS-82]. These are indications that the two-matroid intersection problems are difficult to solve in parallel efficiently although this does not rule out the possibility that it can be done.

The results of this thesis show that perfect speed-up can be achieved for the two-matroid intersection problems. Fast algorithms which run in almost linear time are derived for the general problems. A sub-linear time algorithm is derived for the unweighted matching problem. Although it still remains to be shown whether sub-linear time can be achieved for the general problems, the results of the thesis demonstrate the power of parallel processing and the appropriateness of the chosen model.

(2) The second objective is to obtain some general strategies for deriving parallel algorithms which can be used for other problems. There are at least three techniques which are applicable to most of the problems in this thesis.

The first one is "recursive doubling" [FW-78]. With this technique, we increase the size of search or temporary solution set by double at each iteration, so that a logarithmic time complexity can be achieved. This technique arises in many places in this thesis and we believe that it is useful in the design of parallel algorithms in general.

The second technique is a parallel breadth-first search which solves the shortest-paths problem in logarithmic time. It is used in all the fast algorithms in this thesis.

The third technique is to apply "Brent's Theorem" to decrease the number of processors used without increasing the parallel time. This method is adopted from [SV1-82]. It is applied in all problems in this thesis to achieve perfect speed-up. It should also be useful for other parallel algorithms.

(3) The third objective is to observe the impact on parallel algorithms of the parallel model. Since the chosen problems are very demanding of the model of parallelism, they have actually led to the introduction of two new operations for the model. Also, they have indirectly led to the suggestion of a more powerful model, the "concurrent critical section model", which could be very useful for parallel algorithm design for other difficult problems.

1.2. Definitions and Examples

In the following, definitions and examples of matroid problems are given. Matroids are special *subset systems* where subset systems are defined as follows:

Definition 1.1 A *subset system* (*independence system, hereditary set system*) $S = (E, \mathbf{I})$ is a finite set E and a collection I of subsets of E such that

1.1.1 $\emptyset \in \mathbf{I}$

1.1.2 If $X \in \mathbf{I}$ and $Y \subseteq X$ then $Y \in \mathbf{I}$ (*hereditary* property of independence).

For every subset system, we can define a related problem as follows:

Definition 1.2 A *combinatorial optimization problem associated with* S is: given a weight $w(e) \geqslant 0$ for each $e \in E$, find an element of $\mathbf{I}$ with largest weight.

The following are two examples of combinatorial optimization problems associated with subset systems.

Example 1.1 Given a graph $G = (V, E)$ with non-negative edge weights, find a *maximum weight spanning forest* of G. E is the *ground* set (i.e., the finite set of Definition 1.1) and $\mathbf{I}$ is the collection of cycle-free subsets of E.

Example 1.2 Given a digraph $D = (N, A)$ with non-negative arc weights, find a subset $B \subseteq A$ of largest weight such that no two arcs of B have the same head.

It can be proved that these two optimization problems are both solved by the well known greedy algorithm.

Algorithm 1.1 : Greedy Algorithm

($S = (E, \mathbf{I})$ is a subset system. X will be the solution.)

```
$X \leftarrow \emptyset$        $A \leftarrow E$
while $A \neq \emptyset$
    choose element $e \in A$ with largest weight
    $A \leftarrow A - \{e\}$
    if $X \cup \{e\} \in \mathbf{I}$ then $X \leftarrow X \cup \{e\}$
```

The two given examples are in fact examples of optimization problems based on matroids. In general any subset system whose optimization problems can be solved by the greedy algorithm is a matroid. Thus, *matroid* can be defined algorithmically as follows (see [PS-82] pg. 285).

Definition 1.3 Let $M = (E, \mathbf{I})$ be a subset system. M is a *matroid* if the greedy algorithm correctly solves any instance of the combinatorial optimization problem associated with M.

Matroid is among the very few mathematical structures which have this interesting relationship with algorithms. We may also define matroid by adding a third axiom to the definition of subset system.

Definition 1.4 A *matroid* $M = (E, \mathbf{I})$ is a structure in which E is a finite set of *elements* and $\mathbf{I}$ is a family of subsets of E such that

1.4.1 $\varnothing \in \mathbf{I}$

1.4.2 If $X \in \mathbf{I}$ and $Y \subseteq X$ then $Y \in \mathbf{I}$

1.4.3 If $U, V \in \mathbf{I}$ with $|U| = |V| + 1$ then there exists an $x \in U - V$ such that $V \cup \{x\} \in \mathbf{I}$.

It can be verified that the independence systems in the two examples above both satisfy condition 1.4.3 in Definition 1.4 and are therefore matroids. In fact, they are examples of two different types of matroids, namely graphic matroids and partition matroids. We shall see more of these matroids in later chapters.

Definition 1.5 Let E be the set of edges of a graph and let $\mathbf{I}$ be the collection of cycle-free subsets of E. Then $(E, \mathbf{I})$ is a *graphic matroid* or *cycle matroid* of a graph.

Definition 1.6 Let E be a finite set of objects, let Π be a partition of E into m disjoint blocks $B_1, B_2, \ldots, B_m$, and let $d_1, d_2, \ldots, d_m$ be positive integers. Then $(E, \mathbf{I})$ is a *partition matroid* where $\mathbf{I} = \{X : X \subseteq E, |X \cap B_i| \leqslant d_i ; 1 \leqslant i \leqslant m\}$. (In this thesis, we will assume the special case of $d_i = 1$, $1 \leqslant i \leqslant m$ unless otherwise specified.)

The following terminology will be used widely in this thesis.

Definition 1.7

Let $M = (E, \mathbf{I})$ be a matroid.

1.7.1 Subsets in $\mathbf{I}$ are called *independent sets*. All other subsets of E are *dependent sets*.

1.7.2 A *base* of M is a maximal independent subset of E.

1.7.3 A *circuit* of M is a minimal dependent subset of E.

1.7.4 The *rank function*, ρ, of M is defined as follows : if A is any subset of E, then $\rho A = \max\{|X| : X \subseteq A, X \in \mathbf{I}\}$. (I.e., the rank of A is the cardinality of a maximal independent subset of A.) The rank of M, denoted ρM, is the rank of E.

1.7.5 A *span* of $A \subseteq E$ is a maximal superset S of A satisfying $\rho S = \rho A$.

Next we shall introduce the matroid intersection problems. Note that there is only one version of the single-matroid problems, namely to find a maximum weight independent set. The maximum weight independent set will also be of maximum cardinality. However, this is not true for the intersection of two or more matroids. Hence, we have two versions of these problems, one concerns the search for a maximum weight intersection, the other concerns the search for a maximum cardinality intersection.

Definition 1.8 Let E be a finite set, $w : E \to \mathbf{R}^+$ a nonnegative weighting function on E and let $M_i = (E, \mathbf{I}_i)$ $1 \leqslant i \leqslant n$ be n matroids over E. The *maximum weight matroid intersection problem* is the problem of finding a set $I \in \bigcap_{i=1}^{n} \mathbf{I}_i$ of maximum weight. The special case of the maximum weight matroid intersection problem when $w(e)=1$ for all $e \in$ E is known as the *maximum cardinality matroid intersection problem.*

This thesis will focus on two-matroid intersection problems. Parallel algorithms will be designed for the general problems and also for two special cases, the bipartite matching problems and the directed spanning tree problems.

Example 1.9 : Bipartite Matching Problems.

Let G=(X,Y,E) be a bipartite graph, so that X and Y are two disjoint sets of vertices and E is a set of edges in X×Y. A subset I ⊆ E is called a matching for G if no two edges in I are incident to the same vertex. The *cardinality bipartite matching problem* is to find a matching of G of maximum cardinality. We formulate this problem as a maximum cardinality matroid intersection problem as follows. Let $X=\{x_1, \cdots, x_n\}$ and $Y=\{y_1, \cdots, y_m\}$, and let $\Pi_i=\{(x_i, y) \in E \mid y \in Y\}$ for $1 \leqslant i \leqslant n$ and $\Gamma_j=\{(x, y_j) \in E \mid x \in X\}$ for $1 \leqslant j \leqslant m$. Then the Π_i 's (respectively the Γ_j 's) partition E into blocks such that any two edges of E are in the same block iff they are incident to the same vertex in X (respectively Y). Let M_1 =(E, $\mathbf{I}_1$) and M_2= (E, $\mathbf{I}_2$) be the *partition matroids* determined by the partitions of E induced by the Π_i 's and the Γ_j 's respectively. Hence a set I ⊆ E is in $\mathbf{I}_1$ iff no two edges in I are incident to the same vertex in X. Similarly I ∈ $\mathbf{I}_2$ iff no two edges in I are incident to the same vertex in Y. Thus $\mathbf{I}_1 \cap \mathbf{I}_2$ is exactly the set of matchings of G and so a solution to the bipartite matching problem is given by a set I ∈ $\mathbf{I}_1 \cap \mathbf{I}_2$ of maximum cardinality.

The *weighted bipartite matching problem* is defined similarly except that each of the edges in E has a weight and the problem is to find a matching of G of maximum weight. The corresponding matroid problem will be the weighted two-matroid intersection problem for matroids M_1 and M_2.

Example 1.10 : Directed Spanning Tree Problem

Let G=(V,A) be a weighted directed graph with a distinguished vertex v of indegree 0. We wish to find a maximum weight spanning tree of G rooted at v. Let M_1=(A,$\mathbf{I}_1$) be the *graphic matroid* of G where we agree to ignore the direction of the arcs. Thus I ∈ $\mathbf{I}_1$ iff I is a set of edges which contains no cycle. Let V=$\{v_1, \cdots, v_n\}$ and let (v_i, v_j) denote an arc from v_i to v_j. Then the sets $\Pi_i=\{(v, v_i) \in A \mid v \in V\}$; $1 \leqslant i \leqslant n$ partition A into disjoint subsets. Let M_2=(A,$\mathbf{I}_2$) be the *partition matroid* over A induced by the Π_i's. A set I ∈ $\mathbf{I}_2$ only if no two arcs enter the same vertex. Thus $\mathbf{I}_1 \cap \mathbf{I}_2$ is the set of all subgraphs of G which are directed forests. A spanning tree is a subset I ∈ $\mathbf{I}_1 \cap \mathbf{I}_2$ such that $|I|=n-1$.

The three-matroid intersection problems are NP-complete. An example is the Directed Hamilton Cycle problem [GJ-79]. We do not expect these problems to be solved efficiently even on parallel computers. Approximation algorithms will be needed to solve them efficiently within some error range.

Example 1.11 Directed Hamilton Cycle

Let G=(V,A) be a directed graph. We wish to find a directed Hamilton cycle in G. This problem can be realized as a maximum cardinality problem for the intersection of three matroids. Let $n = |V|$ and create a new directed graph G'=(V',A') where

$V' = V \cup \{v_{n+1}\}$ and $A' = \{(v_i, v_j) \in A \mid 1 \leq i \leq n, 2 \leq j \leq n\} \bigcup \{(v_i, v_{n+1}) \mid 1 \leq i \leq n, (v_i, v_1) \in A\}$

Let $M_1 = (A', I_1)$ be the *graphic matroid* of G'. Let $\Pi_i = \{(v, v_i) \in A' \mid v \in V'\}$ and $\Gamma_i = \{(v_i, v) \in A' \mid v \in V'\}$ for $1 \leq i \leq n+1$. Then the sets Π_i (respectively Γ_i) partition A' into blocks such that arcs in different blocks enter (respectively leave) different vertices of G'. Let $M_2 = (A', I_2)$, $M_3 = (A', I_3)$ be the *partition matroids* over A' corresponding to the partitions induced by the Π_i's and the Γ_i's respectively. Then a subset $I \subseteq A'$ is in $I_1 \cap I_2 \cap I_3$ iff it is cycle free (so it is in I_1), it does not contain two arcs going into the same vertex (so it is in I_2) and it does not contain two arcs going out of the same vertex (so it is in I_3). In particular a subset $I \in I_1 \cap I_2 \cap I_3$ of maximum cardinality $|I| = n$ is a path from v_1 to v_{n+1}. By finding such an I we may then easily construct the desired Hamilton cycle by replacing the edge $(v_i, v_{n+1}) \in I$ by (v_i, v_1).

1.3. Previous Work

A survey of parallel algorithm design can be found in [Vi-83]. Algorithms and data structures developed to solve graph problems on different models of parallel computers are surveyed in [QD-84]. In relation to the matroid problems, a parallel greedy algorithm has been designed in [Co-83]. Different versions of parallel algorithms for the minimum spanning tree problem, which is a single matroid problem, have been designed, and we shall give a review at the end of chapter 3. The unweighted matching problem can be transformed into the max-flow problem, which has a parallel algorithm [SV-82]. Dekel and Sahni [DS-82] have developed an algorithm based on a SIMD (Single instruction stream multiple data stream) model to find the maximum cardinality matching of a *convex* bipartite graph. A bipartite graph is *convex* if there is an ordering of the vertices $X=\{x_1,x_2,\ldots,x_{|X|}\}$ and $Y=\{y_1,y_2,\ldots,y_{|Y|}\}$ such that for all triplets i,j,k with $i<j<k$, $(x_h,y_i)\in E$ and $(x_h,y_k)\in E$ implies $(x_h,y_j)\in E$. The algorithm in [DS-82] takes $O(\log^2 n)$ time using $O(n)$ processors, where n is the number of vertices.

The all-pairs and single-source shortest-paths problems are sub-problems of the two-matroid intersection problem and the bipartite matching problem respectively. Parallel algorithms have been designed to solve the shortest-paths problem on different models. A list of these algorithms is given near the end of chapter 5.

There exist polynomial time sequential algorithms for all the problems in this thesis. For the cardinality and weighted two-matroid intersection problems, there are algorithms [La-76],[PS-82] based on augmenting sequences (see chapter 4) and algorithms [La-76], [PS-82], [Fr-81] based on the primal-dual method of linear programming. The same holds for the bipartite matching problems ([La-76] [PS-82]). The directed spanning tree algorithm can be found in [La-76] or [Ta-77].

1.4. Overview

In the next chapter, parallel models are examined and the model used in this thesis is described. In particular, two new operations, "write-max" and "write-min", are introduced. Also, a new model called the "concurrent critical section" model is proposed and shown to be a more powerful model than the existing ones.

Chapter 3 contains both sequential and parallel greedy algorithms for the single-matroid problem and also lists the best algorithms known for the minimum spanning tree problem.

Chapter 4 describes the sequential algorithms and derives parallel algorithms for the general cardinality and weighted two-matroid intersection problems. Chapters 5 and 6 describe the sequential algorithms and derive parallel algorithms for the special cases of bipartite matching and directed spanning tree problems respectively. Conclusions and open problems are given in Chapter 7. The results from chapters 4,5, and 6 are summarized in the Appendix.

2. PARALLEL MODELS

Before designing a parallel algorithm, we must define the abstract model of parallel computation which specifies the "design space" in which we work. There are many different parallel models of computation. We shall list some of them and justify the choice of model in this thesis.

2.1. Some Common Models

In this thesis, we are concerned with parallel models consisting of a "tightly coupled" collection of parallel processors working together to solve a terminating computational problem. There are several criteria by which we can classify parallel models.

(1) The number of processors may be fixed or unbounded. In this thesis, the number of processors will be assumed to be polynomial in the number of inputs in the problems.

(2) We may classify the models according to the pattern of processor and memory intercommunication. Preparata and Vuilemin [PV-79] have distinguished two broad categories of such parallel models.

(i) Models based on a fixed connection network of processors

These models assume that only graph theoretically adjacent processors can communicate in a given step. Kung [Ku-82] has focused on the design of parallel algorithms that conform well to "systolic" architectures which lay out well in two dimensions. These systolic systems are examples of models based on a fixed connection network of processors.

The Ultracomputer of Schwartz [Sc-80], mesh-connected processors such as Illiac IV , and the cube-connected cycles of [PV-79] also belong to this category.

The structure of this kind of model dictates ~~that~~ a problem be decomposed into identical subtasks which communicate among each other in some regular fashion. Hence it is not suitable for problems with many data dependent decisions [NYU-83].

(ii) Models that are based on the existence of global or shared memory

The models in this category consist of processors which have access to a shared memory. Several models are defined which differ in whether or not they allow concurrent read and write operations to the shared memory, and, if allowed, how write conflicts are resolved [BH-82].

1) PRAC [LPV-81] - concurrent read and concurrent write are not allowed.

2) CREW PRAM [FW-78] - concurrent read allowed , concurrent write not allowed.

3) CRCW PRAM - concurrent read allowed, the allowance of concurrent writes can differ as follows:

(i) in [SV-80], concurrent write is allowed only if all processors are trying to write the same thing.

(ii) An arbitrary processor is allowed to write.

(iii) General CRCW PRAM ([Go-78]) - the lowest numbered processor is allowed to write.

The model used in this thesis is a general CRCW PRAM. A proper definition of this model is given in the next section.

(3) In [QD-84], a distinction between SIMD (single instruction stream multiple data stream) and MIMD (multiple instruction stream multiple data stream) models is made. SIMD is taken to be synonymous with "processor array".

(i) In a SIMD model, the processors may communicate with each other via a shared memory (SM) or some kind of network, such as a mesh-connected network (MC), a perfect shuffle network (PS), or a cube-connected cycles network (CCC). For example, in a SIMD-MC model, the processors are arranged in a q-dimensional lattice, and communication is allowed only between neighboring processors. Most supercomputers of the current generation belong to this category. They are composed of vector pipelines which are multiple processors each executing the same instruction [St-80]. For example, the ILLIAC IV computer belongs to the SIMD-MC category. It is an array processor composed of 64 identical processing elements, organized as an 8x8 array, which synchronously execute the same instruction (possibly operating on different data).

Although our algorithms are designed for the CRCW PRAM models, they also works for the SIMD models given the same kind of allowances in memory access.

(ii) The MIMD models may also differ in the processor intercommunication pattern. For example the MIMD-TC(for tightly coupled) model assumes that all processors work through a central switching mechanism to reach a global memory. Our chosen model belongs to this category.

(4) The processors may be synchronous or asynchronous. Here we shall make the distinction between different types of synchronization.

If the processors or RAM's (random access machines) are "clock-synchronized", it means that they are working under the control of a global clock. All tightly coupled systems are clock-synchronized.

There is a second kind of synchronization:
A "clock-synchronized" parallel model may be "asynchronous". It will be an MIMD-TC model according to the previous classification. This means that the processors are working independently most of the time. They may be executing different instructions and there will be some points at which they have to be "synchronized".

An example of such a model is given in [SV-81]: "There is a universal clock to the program that ticks every time unit and each processor can perform one and only one elementary operation between two ticks. A starting time will be assigned to some of the instructions. The execution of such an instruction must start exactly in the starting time assigned to it. This enables us to achieve synchronization whenever necessary.
In our model, the synchronization is defined in a different way : at a point of synchronization, each processor has to wait until all other processors have arrived at this point before they can continue with their processing.

2.2. The General CRCW PRAM

The CRCW PRAM (concurrent-read concurrent-write parallel random access machine) model is a MIMD machine model. The following definition is adapted from [Vi-83]. We have changed the word "synchronously" in the original definition into "clock-synchronously" to distinguish between the two types of synchronization.

Definition 2.1 : The general CRCW PRAM model has p RAM's operating "clock-synchronously" in parallel. Each RAM is a standard uniprocessor model having its own large local random-access memory and has instructions for typical arithmetic and boolean operations and for reading and writing its local memory. The RAM's also have access to a shared memory of size m. Each RAM has instructions for reading from and writing into the common memory. Several processors may read simultaneously from the same memory location. If more than one processor attempts to write into the same location in the common memory at the same time, the lowest numbered processor succeeds. The CREW PRAM (concurrent-read exclusive write PRAM) is similarly defined except that simultaneous writes into the same location are not allowed.

The work of Cook and Dwork [CD-82] implies that a CRCW PRAM is more powerful than a CREW. It is easy to see that any program that runs on the CREW PRAM model will also run on the CRCW PRAM model within the same time and space bounds. A simulation of the CRCW PRAM model by the CREW PRAM model will be given later and it will add a logarithmic factor to the time complexity.

2.3. The Fetch-and-Add PRAM Model

This is a modification of the CRCW PRAM. Let A be a common memory address and let r be the value of a local register of processor P. The Fetch-and-Add (F&A) instruction is defined as follows : If processor P performs a F&A(A,r) and no other processor performs at the same time an instruction that relates to address A, then the contents of A are transmitted to processor P and address A is assigned the value A+r. If several processors simultaneously perform F&A instructions that relate to A, the result is defined to be the same as if these instructions are performed serially in some (unknown) order. The F&A PRAM is a CRCW PRAM that allows these F&A instructions.

2.4. The NYU-Ultracomputer

The Fetch-and-Add model will be realized by the NYU-Ultracomputer. The NYU-Ultracomputer is a shared memory MIMD parallel machine composed of thousands of autonomous processing elements. It uses an enhanced message switching network with the geometry of the Omega network of Lawrie [La-75] to implement efficiently the fetch-and-add synchronization primitive. The Omega network consists of $N \log N$ 2x2 switches and connects N processing elements (PE's) to N memory modules (MM's). The MM's are standard off-the-shelf memory chips. The PE's are slightly custom designed for the F&A operation. Each PE is attached to the network via a processor network interface (PNI) and each MM is attached via a memory network interface (MNI). Figure 1 gives a block diagram of the machine. Shared memory access time in this machine has latency time that is logarithmic in N [NYU-83].

Figure 2 shows an example of an omega network with $N = 8$. The small circles on the left are the PE's and those on the right are the MM's. Suppose a fetch-and-add(001, 3) command is made by processor 101 to reference memory module 001. the dotted line shows the path of this reference through the network. If a fetch-and-add(001, 4) command is made by processor 111 and the two references meet at switch A, then the two references are combined at A and the instruction fetch-and-add(001, 3+4) is passed through the remaining dotted path.

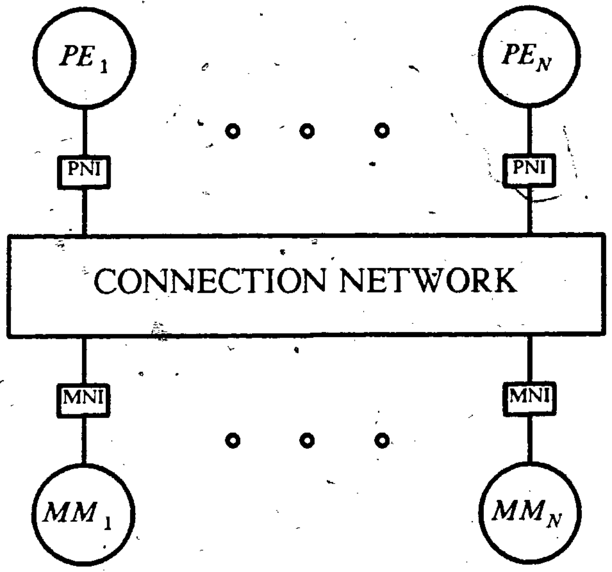


Fig 1. Block diagram for NYU Ultracomputer

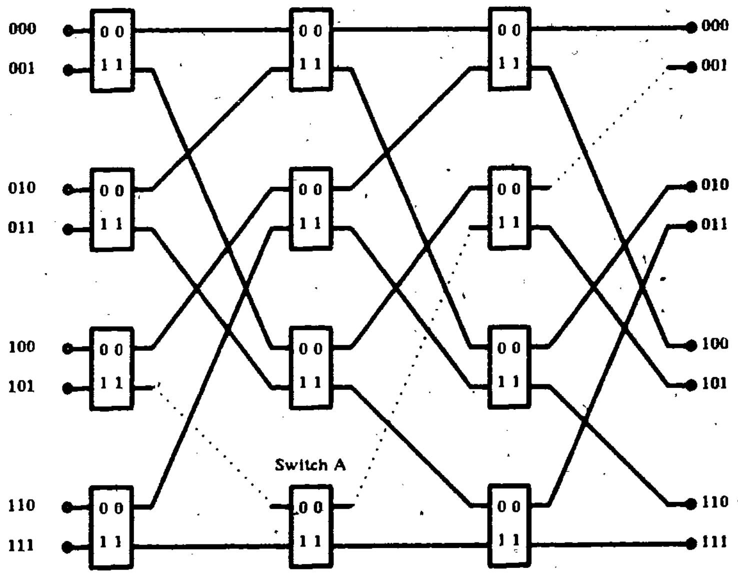


Fig 2. Omega-network(N = 8)

2.5. The write-max and write-min operations

The Fetch-and-Add operation is one special case of a more general fetch-and-Ø operation which may be used as the sole primitive for accessing shared memory [NYU-83]. Let V be a common memory address and e be the value of a local register of a processor. fetch-and-Ø(V, e) will fetch the value in V and replace it with Ø(V,e) where Ø is an associative and commutative function. (If Ø(a,b) = a + b then we have the fetch-and-add operation.)

In [NYU-83] the authors remark that the fetch-and-add operation has proved to be a sufficient coordination primitive for all the highly concurrent algorithms developed to date. However, during the design of parallel algorithms for matroid problems, two new operations arise naturally and they prove to be very useful both conceptually and in terms of improving the time efficiency. These are the "write-max" and "write-min" operations. These two operations are also special cases of the general fetch-and-Ø operation. They are defined as follows :

Definition 2.2 : If processor P performs a write-max(A,e) and no other processor performs at the same time an instruction that relates to address A, then the value of e will be written into A if and only if e is greater than the content of A. If k processors $P_1, P_2, \ldots, P_k$ simultaneously perform instructions write-max(A,e_1), write-max(A,e_2),... write-max(A,e_k) respectively, then address A will be assigned the value of max{ $e_1, e_2, \cdots e_k$, C(A) } where C(A) is the original content of A. No value is returned to the processors.

write-min(A,e) is defined in the same way except that the minimum among the values $e_1, e_2, \cdots e_k$ and C(A) is assigned to address A instead of the maximum.

Implementation of the New Operations

The implementation of fetch-and-add is described in [NYU-83] and a machine model with an omega network is suggested. We shall assume the same framework in the implementation of write-max or write-min. In fact, only very simple modifications to the machine will be sufficient. We shall only describe the implementation for write-max since write-min can be similarly implemented.

To implement write-max, the switches will be enhanced to permit the network to combine write-max instructions with the same efficiency as it combines loads and stores. We include comparators in the switches and also in the memory network interfaces (MNI's). When two write-max's referencing the same shared variable, say write-max(X,e) and write-max(X,f) meet at a switch, the switch computes max{e,f} and transmits the combined request write-max(X, max{e,f}). When a write-max(X,e) request reaches the MNI associated with the MM containing X, the content of X and the transmitted e are sent to the MNI comparator, and the greater value is stored in X.

Since we are not interested in getting a return value and the comparator is no more complicated than an adder, we see that the write-max operation is no harder to implement than the fetch-and-add operation.

Next we must consider the combination of write-max with other operations referencing the same location and formulate rules about the validity of such concurrent references.

(1) write-max and load (read) :

There is no conflict between these two; a combined request of fetch-and-write-max(X,e) is transmitted which means that C(X), the content of X, will be returned to the processor doing the load and max(e,C(X)) will be written to X.

(2) write-max(X,e) and store(X,f) :

We may do any one of the following:

(i) transmit the store command (assume it comes second)

(ii) transmit store(X, max{e,f}) (assume the write-max command comes second)

(iii) make this combination illegal

(3) write-max and F&A :

There is conflict between these two instructions; there is no way to combine these two at a switch to guarantee that it preserve a serial order. For example, suppose we have a write-max(X,a) and a F&A(X,b) to be combined at a switch, and suppose the current content of X is c. The following are all of the possible outcomes.

(i) If $a \geqslant b+c$ then the value of X will become $a+b$ if the write-max comes first or a if the F&A comes first. To conform to some serial order of the instructions, we may combine the two into a write(X, $a+b$) or a write(X,a) instruction.

(ii) If $b+c \geqslant c \geqslant a$ then the value of X will become $b+c$ independent of the whether the write-max or the F&A comes first. Hence we should combine the two into a F&A(X,b) instruction.

(iii) If $b+c \geqslant a \geqslant c$ then the result of X will be $a+b$ if the write-max comes first, and $b+c$ if the F&A comes first. Hence we may combine the two into either a write(X,$a+b$) or a F&A(X,b).

We see that we can distinguish between these cases only if we know the value c. However, we have no knowledge about the content of X at the switch. If we mix up these cases, we cannot guarantee that the result is equivalent to the result of some serial ordering of the instructions. Hence, we have to make this combination illegal.

(4) write-max and write-min : There is a conflict between these two instructions, and we make this combination illegal.

In fact, in the applications of write-max or write-min in the algorithms in this thesis, there is no incidence of concurrency with the store or F&A instructions. With no loss of generality, we make all such concurrency illegal, which may also be a cleaner design.

Now we are ready to define the model that is used in this thesis.

Definition 2.3 : The *write-max/write-min F&A PRAM* model is a F&A PRAM which allows the write-max and write-min operations with the above rules for concurrent references.

2.6. The Concurrent Critical Section Model

In this section, we introduce a new parallel model called the "concurrent critical section model". None of the algorithms in later chapters use this model, but we believe that the concurrent critical section model is sufficiently interesting to justify its inclusion here. We will first state the motivation for introducing this model and show why it is more powerful than other existing models. Then we will show that this model is no more complicated or difficult to simulate by the CREW PRAM model than the CRCW or F&A CRCW PRAM models. To do this we will first describe the simulation of concurrent writes and Fetch-and-add instructions on a CREW PRAM model. Then we will describe the simulation of concurrent critical sections on the CREW PRAM model and show that it has the same time and space complexities as the first two simulations of concurrent writes and Fetch-and-add's.

We define a critical section to be a section of a parallel algorithm in which more than one processor may be accessing one or more common variables at the same time, and the result of these references must be the same as if the critical sections for these processors are done in some serial order. When we say that the general CRCW PRAM model allows concurrent writes, we are actually saying that it takes O(1) time for several processors to execute a critical section containing one concurrent write instruction.

However, if a critical section contains instructions referencing more than one variable, then the general CRCW model will not be able to resolve it in constant time. For example, a common critical section for checking a semaphore is

```
Test-and-set(V)
        { temp ← V
          V ← true }
        return temp
```

All processors executing this critical section are competing for a certain semaphore V. Only one processor can be allowed to get the semaphore. The value of V is initially false. In the critical section a processor will read the value of V into a local variable temp, and then set the value of V to true. Hence only one processor will receive the value of false in temp and it gets the semaphore.

The CRCW model cannot solve this problem in constant time although it can simulate it in logarithmic time (as described in the next section). The fetch-and-add operation is introduced to resolve critical sections like this. For example, the above critical section for checking a semaphore can be replaced by a single instruction

fetch-and-add(V,1).

Assuming 0 is the initial value of V, the processor that gets a zero as a return value obtains the semaphore.

However, if the critical section is more complicated, some other operations may be needed. For example if all processors are writing to a variable and we want the one that writes the maximum value to succeed, then we need a "write-max" operation. Also, we may want to reference and modify more than one global variable in a critical section. To address this issue, we introduce a model that assumes constant time to resolve critical sections in general. We call this the "concurrent critical section" model.

Definition 2.4 : The *concurrent critical section* model has p RAM's (random access machines) operating clock-synchronously in parallel. Each RAM has instructions for reading from and writing into a common memory of size m. If several processors simultaneously execute critical sections which compete for access to some global variables, the result is the same as if these critical sections are performed serially in some (unknown) order and the time required is the time for executing any one of the critical sections.

Lemma 2.5: The CRCW PRAM , the F&A PRAM, and the write-max/write-min F&A PRAM models are special cases of the concurrent critical section model.

Proof : Concurrent write is equivalent to a single instruction critical section:

```
{ write(X,e) }
```

The fetch-and-add(X,e) instruction is equivalent to the critical section

```
{ read X
  write(X,e+C(X)) }
where C(X) is the content of X
```

The write-max(X,e) instruction is equivalent to the critical section

```
{ if e > C(X) then
     X ← e     }
```

The write-min(X,e) instruction is equivalent to the critical section

```
{ if e < C(X) then
     X ← e     }
```

□

In the next three sections, we compare this new model with the CRCW PRAM and F&A PRAM models. Since the CREW model is widely accepted as a realistic model, we will simulate each of the three models using the CREW model. We shall see that all three simulations have the same time and space bounds.

2.6.1. Simulating Concurrent Writes on the CREW PRAM Model

At any point that there are possible concurrent writes in a given algorithm, we synchronize all processors and direct these writes to temporary locations. The simulation uses the recursive doubling technique. Synchronization is necessary and every processor must be used.

Suppose there are m processors and there is a concurrent write to location X. We create temporary variables T(i) and M(i) for each processor P_i. Instead of writing to X, each P_i writes to T(i). Next, the processors are synchronized (the method of synchronization will be given later). Now, every P_i with $i \equiv 0 \; mod \; 2^2$ checks T(i) and T($i+1$) and combines them according to the serial order of the CRCW model (e.g. the lowest numbered processor succeeds). The combined result is written to T(i).

The processors are synchronized again. This time every P_i with $i \equiv 0 \; mod \; 2^2$ checks T(i) and T($i+2$) and combines them into T(i). In general, at the j-th synchronization, every P_i with $i \equiv 0 \; mod \; 2^j$ checks T(i), T($i+2^{j-1}$) and combines them into T(i).

The above process is iterated $\left\lceil \log_2 m \right\rceil$ times and the final combination of all writes is written into X.

The method of synchronization is as follows: At the j-th synchronization, each P_i with $i \equiv 0 \; mod \; 2^j$ sets M(i)$\leftarrow j$. At the next stage, a processor P_i with $i \equiv 0 \; mod \; 2^{j+1}$ will not proceed until M(i)$=j$ and M(i+2^{j-1})$=j$.

It is easy to see that the simulation takes O(log$m$) time and O(m) memory space for each concurrent write.

2.6.2. Simulation of Concurrent F&A on the CREW PRAM We shall use O(m) memory space and O(log$m$) time to simulate the network of the NYU-Ultracomputer. The technique is similar to the previous simulation. Synchronization is done in the same way. However, since we have to remember some temporary results at each level of the network, we have to create temporary memory locations for these levels as well. An m x log$m$ array, T, is used where T(i,j) is used at the j-th synchronization by processor P_i.

For example, if fetch-and-add(X,e) and fetch-and-add(X,f) arrive at T(a,j) and T(b,j) and are to be combined at the next level, then at T(a,j+1) the instruction will be fetch-and-add(X,e+f). T(a,j) has a residual value of e and T(b,j) has a residual value of e+f.

After the final write into X, the previous value at X is returned through the T(i,j)'s as follows:

> The value received at T(a,j), say Y, is added to e and Y+e is returned to the previous level to T(a,j-1) and T(a/2,j-1). If Z is the value received at T(b,j), then Z+e+f is returned to the previous level.

It can be seen that at the i-th synchronization, only the T(i,j)'s for $j \equiv 0 \ (\mathit{mod}\ 2^i)$ are used. T can be replaced by a complete balanced binary tree with m leaves. The number of nodes in the tree is $2m-1$ and $O(m)$ memory space is required.

2.6.3. Simulation of Critical Sections on CREW PRAM Model This simulation will require $O(T \log m)$ time and $O(Mm)$ space, where m is the number of processors, T is the time, and M is the number of variables involved in the critical section. A generalization of the mechanism for simulating F&A can be applied here. Instead of dealing with a single variable as in fetch-and-add, we now deal with $M \geqslant 1$ variables, $V_1, V_2, \cdots V_M$. Hence each variable, V_k involved in the critical section will be given a temporary m by $\log m$ array T_k. Synchronization is done exactly as before. Now everything to be done in the critical section is done using the temporary variables and, in $O(T \log m)$ time these critical sections are combined in some serial order consistent with the definition of the model. As in the simulation of F&A, the arrays T_k can be replaced by binary trees of size $2m-1$. Hence the memory required is $O(Mm)$.

From the above results, the proposed model is no harder to simulate with a CREW machine than the CRCW and F&A models. From an implementation point of view, the switches of the network of the NYU-Ultracomputer can be enhanced so that they can combine critical sections.

The new model is definitely at least as powerful as the existing PRAM models since the existing models are special cases of the new model. We expect this new model to be useful in the design of parallel algorithms both conceptually and in terms of efficiency for difficult problems. It is also practical in the sense that implementation is easy. Whether the implementation cost is justified will depend on results in the design of parallel algorithms for this model.

2.7. Parallel Algorithms in this Thesis

The parallel algorithms in this thesis are designed for the write-max/write-min F&A PRAM model (Definition 2.3). (The concurrent critical section model is not used.) The algorithms are written in a program-like fashion using common keywords such as "for", "while", "if..then..else", and so on. Most parts of the algorithms will be simultaneously executed by many processors. We shall give indices to the processors and call a processor "P_{ij}", where i and j are the indices. The indices may appear in the section of the algorithm which P_{ij} executes. The sign '$\leftarrow^{*}$' means concurrent write. The sign '$\leftarrow^{**}$' means concurrent write of possibly different values to a location.

Most of the time, the processors are working asynchronously. At points where synchronization is needed, a <*synchronize*> statement will appear. This means that each processor must wait at this point until all other processors have finished their work to this point. Then all processors can proceed. This synchronization can be done using the fetch-and-add instruction as follows.

Let p be the number of processors. Three global variables $SYNC_0$, $SYNC_1$ and $SYNC_2$ are used. Initially, $SYNC_0$, $SYNC_1$, and $SYNC_2$ are 0. Each processor has a local variable i which has initial value of 0. When a processor comes to a synchronization point, it performs a fetch-

and-add($SYNC_i$, 1) operation. If the return value is p, then the processor changes $SYNC_{(i+2) \, mod \; 3}$ to 0 and changes its local variable i to $(i+1) \; mod \; 3$. It then proceeds with its work. If the return value is not p, then the processor will keep reading the value of $SYNC_i$ until its value becomes p. It then changes its local variable i to $(i+1) \; mod \; 3$ and proceeds with its work.

3. GREEDY ALGORITHMS FOR SINGLE-MATROID PROBLEMS

We mentioned in Chapter 1 that single-matroid problems can be solved by the greedy algorithm. In this chapter, we describe a parallel greedy algorithm and list the best known parallel algorithms for the special case of the minimum spanning tree problem.

3.1. The General Single-Matroid Problem

Before developing a parallel greedy algorithm, we should look at the serial algorithm described in Chapter 1 more closely.

Algorithm 1.1 : Greedy Independence Algorithm

```
        ($M = (E, I)$ is a matroid.   X will be the solution.)
1.      $X \leftarrow \varnothing$          $A \leftarrow E$
2.      while $A \neq \varnothing$
3.              choose element $e \in A$ with largest weight
4.              $A \leftarrow A - \{e\}$
5.              if $X \cup \{e\} \in I$ then $X \leftarrow X \cup \{e\}$
```

If $|E| = m$ then the loop is executed $O(m)$ times. To implement line 3, E could be made into a heap before entering the loop and line 3 would then be $O(\log m)$. Alternately, E could be sorted before entering the loop and line 3 would be $O(1)$. In both cases, the total contribution is $O(m \log m)$.

The greedy algorithm above is based on the *independence axioms* of Definition 1.4. The time to test a subset for independence in line 5 will depend on the structure of M. In general, if $c(m)$ is the time to test for independence in M, then the algorithm is $O(m\ c(m)) + O(\text{sorting})$.

The algorithm above appears to be inherently sequential: an element cannot be tested for inclusion in X (in line 5) until membership in X has been tested for all elements with larger weights. However, there are ways to find greedy solutions using algorithms based on other properties of matroids. The following greedy algorithm based on the rank function (Definition 1.7) leads naturally to a parallel greedy algorithm.

3.1.1. Greedy Rank Algorithm

Algorithm 3.1 : Greedy Rank Algorithm

```
        ($M = (E, I)$ is a weighted matroid with $|E| = m$.  $X$ will be the solution.)
1.      $X \leftarrow \emptyset$
2.      Sort $E$ by non-increasing weight
3.      for $i \leftarrow 1, \ldots, m$
4.          if $\rho(e_1, e_2, \ldots, e_i) > \rho(e_1, e_2, \ldots, e_{i-1})$ then $X \leftarrow X \cup \{e_i\}$
```

If we let $r(M)$ be the time necessary to compute the rank function of matroid M, the time complexity of the loop at line 3 is $O(m\ r(M))$. Thus the total time complexity is $O(m\ r(M))$ + O(sorting). The difference in time complexity between this algorithm and the Greedy Independence Algorithm is the difference between the time for determining rank and independence and this depends upon the matroid given.

Note that our new algorithm, while not parallel, no longer appears to be inherently sequential. We now consider a parallel greedy algorithm from [Co-83].

Algorithm 3.2 : Parallel Greedy Rank Algorithm [Co-83]

```
        m processors $P_i$, $1 \leqslant i \leqslant m$, execute the following code in parallel
        ($M = (E, I)$ is a weighted matroid with $|E| = m$.  $X$ will be the solution.)
1.      $X_i \leftarrow 0$
2.      Sort $E$ by non-increasing weight  (so that $w(e_1) \geqslant \cdots \geqslant w(e_m)$)
        <Synchronize>
3.      if $\rho(e_1, \ldots, e_i) > \rho(e_1, \ldots, e_{i-1})$ then $X_i \leftarrow 1$  (set $X_i = 1$ if $e_i \in X$)
        <Synchronize>
```

Using m processors, line 1 is O(1), and line 3 is $O(r(M))$ The parallel sort of line 2 requires $O(\log m)$ time and $O(m)$ processors using the method of [Le-84], so the total time is $O(r(M) + \log m)$. The *time-processor product* for this algorithm is $O(m\, r(M) + m \log m)$, giving a perfect speed-up.

The details of the rank computations in line 3 will depend on the structure of M. For some types of matroids, ranks can be computed quickly in parallel. One way to compute ranks in graphic matroids is based on the following.

Fact 3.1 Let $G = (V, E)$ be an undirected graph, and $E' \subseteq E$. Then $\rho E'$ = (number of vertices in $G(E')$) − (number of connected components in $G(E')$), where $G(E')$ is the subgraph of G induced by E'.

Let $|V| = n$ and $|E| = m$. The number of vertices in $G(E')$ can be counted in O(1) parallel time using $\max(n, m)$ processors. Y is a vector of length $\max(n, m)$ and each processor P_i, $1 \leq i \leq \max(n, m)$ executes the following code.

```
$Y_i \leftarrow 0$
$Y_j \leftarrow^* 1$ and $Y_k \leftarrow^* 1$ where $(j, k) = e_i \in E'$
(* The following counts the number of 1's in Y : *)
$COUNT \leftarrow^* 0$
if $Y_i = 1$ then fetch-and-add( $COUNT$, 1 )
```

The number of connected components in $G(E')$ can be computed in $O(\log n)$ parallel time with $n + 2m$ processors using the method in [SV2-82]. The total time for line 3 of the parallel greedy rank algorithm for graphic matroids is therefore $O(\log n)$ using $O(m^2 + mn)$ processors and the total for the entire greedy algorithm is $O(\log n + \log m)$ using $O(m^2 + mn)$ processors. The connected components of a graph can be found in time $O(m + n)$ using one processor and depth-first search. So, a direct sequential implementation of this parallel greedy algorithm would have complexity $O(m^2 + mn)$. The parallel algorithm therefore has an overhead factor of $O(\log m + \log n)$.

Currently, the fastest known parallel algorithm for finding minimum (or maximum) weight spanning trees in a graph is $O(\log n)$ using m processors [AS-83]. So, the parallel greedy algorithm above is not a very good algorithm for graphic matroids. This is probably because the algorithm was derived from an algorithm for arbitrary matroids.

3.2. A Special Case : Minimum Spanning Trees

The minimum spanning tree problem is a special case of the graphic matroid problem. Parallel algorithms have been designed for this problem for different models. The following is a list of the best known algorithms (n is the number of vertices, m is the number of edges, and p is the number of processors):

[AS-83]	uses CRCW and $m \log n / p$ time with $p \leq m$ processors and time-processor product of $m \log n$.
[CLC-82]	uses CREW and n^2/p time with $p \leq n^2/\log^2 n$ processors and time-processor product of n^2
[HV-84]	uses CRCW and $\log n$ time with n^3 processors and time-processor product of $n^3 \log n$
[KR-84]	uses CREW and $m \log n / p$ time with $p \leq m / \log n$ processors and time-processor product of $m \log n$

[AS-83] (resp. CLC-82) derive a Minimum Spanning Forest algorithm (MSF) from a modification of the connectivity algorithm of [SV2-82] (resp. their connectivity algorithm). These MSF algorithms use the same time and number of processors as their respective connectivity algorithms.

4. TWO-MATROID INTERSECTION PROBLEMS

In the previous chapter, we described algorithms for computing a maximum weight independent set of a single matroid. A maximum weight solution is also of maximum cardinality. For the two-matroid intersection problems, the maximum weight set independent in both matroids may not be of maximum cardinality. Hence, two problems can be defined for two-matroid intersections:

(1) Cardinality Two-matroid Intersection Problem : Given two matroids $M_1 = (E, \mathbf{I}_1)$ and $M_2 = (E, \mathbf{I}_2)$ over the same set E, find a maximum cardinality intersection $I \in \mathbf{I}_1 \cap \mathbf{I}_2$.

(2) Weighted Two-matroid Intersection Problem : Given two matroids $M_1 = (E, \mathbf{I}_1)$ and $M_2 = (E, \mathbf{I}_2)$ over the same weighted set E, find a maximum weight intersection $I \in \mathbf{I}_1 \cap \mathbf{I}_2$.

In this chapter we shall deal with these two problems. In both cases, two versions of parallel algorithms are derived. The first version achieves perfect speed-up by applying Brent's Theorem to reduce the number of processors. The second version is a modification of the first version and uses the recursive doubling technique and a logarithmic time breadth-first search. The second version is faster but does not give perfect speed-up. All of these parallel algorithms are based on sequential algorithms from [La-76] which make use of augmenting path methods. For each problem, we describe the sequential algorithm first and then introduce the two versions of parallel algorithms.

4.1. CARDINALITY TWO-MATROID INTERSECTION PROBLEM

The sequential algorithm for the cardinality two-matroid intersection problem is based on "augmenting sequences". Let I be any intersection of the two matroids M_1 and M_2. In the following, if e_i is an element of E, then $I + e_i$ denotes the union of I and $\{e_i\}$ and $I - e_i$ denotes the set I - $\{e_i\}$. These notations are used throughout the thesis.

Now we can construct an augmenting sequence with respect to I as follows. The first element e_1 of such a sequence is such that $I+e_1$ is independent in M_1. If $I+e_1$ is independent in M_2 as well, the sequence is complete; $I+e_1$ will become the next intersection of size one greater than I. Otherwise $I+e_1$ contains a unique circuit in M_2 and we choose e_2 to be an element other than e_1 in that circuit. $I+e_1-e_2$ is clearly independent in both M_1 and M_2. Now we try to find an element e_3 such that $I+e_1-e_2+e_3$ is independent in M_1, whereas $I+e_3$ is not. Such an element is in $sp_1(I)-sp_1(I-e_2)$, where sp_1 denotes span (Definition 1.7) in M_1. If $I+e_1-e_2+e_3$ is independent in M_2, we are done. Otherwise $I+e_1-e_2+e_3$ contains a unique circuit in M_2 and we choose e_4 to be an element in that circuit, and so on.

In other words, the addition to I of the 1st, 3rd, 5th, ... elements preserves independence in M_1, but may create dependence in M_2, whereas the removal of the 2nd, 4th, 6th, ... elements restores independence in M_2. This strategy of adding and deleting elements in turn to and from the intersection I can be done for all possible cases until we arrive at an new intersection or we exhaust the search. If we obtain a new intersection $I+e_1-e_2+....+e_i$ which has size one greater than I, then the search is successful. I is augmented and we start the same search again with the new intersection. In this way, the size of I will be increasing one at a time until it reaches the maximum size. If we exhaust all possibilities and cannot find a bigger intersection, we can stop and return the present intersection as a maximum cardinality intersection. The following definitions for these terms are adapted from [La-76].

Definition 4.1.1 : Let I be an intersection of two matroids $M_1 = (E, \mathbf{I}_1)$, and $M_2 = (E, \mathbf{I}_2)$ Let

$S = (e_1, e_2, ..., e_s)$ be a sequence of distinct elements

where $e_i \in E-I$, for i odd

and $e_i \in I$, for i even

Let $S_i = (e_1, e_2, ..., e_i)$ for $i \leq s$

We say that S is an *alternating sequence* with respect to I if

(1) $I + e_1 \in \mathbf{I}_1$

(2) For all even i, $sp_2(I \oplus S_i) = sp_2(I)$

Hence $I \oplus S_i \in I$

(3) For all odd i > 1, $sp_1(I \oplus S_i) = sp_1(I + e_1)$

Hence $I \oplus S_i \in \mathbf{I}_1$

If, in addition,

(4) $|S| = s$ is odd and $I \oplus S \in \mathbf{I}_2$,

we say that S is an *augmenting sequence* with respect to I.

The following Theorems from [La-76], Chapter 8, prove the validity of the augmenting sequence method described above.

Theorem 4.1.2 : If I is independent in matroid M and $I+e$ is dependent, then $I+e$ contains exactly one circuit in M.

Theorem 4.1.3 : Let I_p, I_{p+1} be intersections of M_1, M_2 with p, $p+1$ elements respectively. Then there exists an augmenting sequence $S \subset I_p \oplus I_{p+1}$ with respect to I_p.

Theorem 4.1.4 : An intersection is of maximum cardinality if and only if it admits no augmenting sequence.

Theorem 4.1.5 : For any intersection I there exists a maximum cardinality intersection I_m such that

$$sp_1(I) \subseteq sp_1(I_m) \text{ and } sp_2(I) \subseteq sp_2(I_m).$$

In the augmenting method as described above, we have to check the circuits that elements form with the current intersection I in either matroid M_1 or M_2. To represent the information about these circuits and facilitate the search for an augmenting sequence, a directed bipartite graph called the *border graph* (BG(I)) will be built with respect to I. The two sets of vertices for the bipartite border graph will correspond to the sets $E-I$ and I. Suppose element $e_i \in E-I$ forms a circuit with I in M_1. Then there will be arcs (e_j, e_i) going from every element $e_j \in I$ in this circuit to e_i. If element $e_k \in E-I$ forms a circuit with I in M_2, then there will be arcs (e_k, e_j) going from e_k to every element $e_j \in I$ in this circuit.

Now, there may exist vertices in the set $E-I$ which have no incoming arcs. These vertices will be called *sources*. Similarly, there may exist vertices in the set $E-I$ which have no outgoing arcs. These vertices will be called *sinks*. Define a *source-sink path* as a path in BG(I) which goes from a source to a sink. We say that a source-sink path admits a shortcut if there exists a shorter source-sink path that goes from the same source to the same sink. It can be shown that the search for an augmenting sequence for I is equivalent to the search for a source-sink path without shortcuts in BG(I). A breadth-first search of BG(I) can be used to find such a path.

The following definitions and lemmas are from [La-76], Chapter 8.

Definition 4.1.6 : For a given intersection I, the *border graph* (BG(I)) is a directed bipartite graph constructed as follows

(i) For each node $e_i \in E - I$ such that $e_i \in sp_1(I)$ there is an arc (e_j, e_i) directed from each $e_j \in C_i^{(1)} - e_i$, where $C_i^{(1)}$ is the unique M_1*-circuit* in $I + e_i$. If $e_i \notin sp_1(I)$, then e_i is a *source* in BG(I).

(ii) For each node $e_i \in E - I$ such that $e_i \in sp_2(I)$ there is an arc (e_i, e_j) directed to each $e_j \in C_i^{(2)} - e_i$ where $C_i^{(2)}$ is the unique M_2*-circuit* in $I + e_i$. If $e_i \notin sp_2(I)$, then e_i is a *sink* in BG(I).

Definition 4.1.7 : Suppose that S is a source-sink path in BG(I) and S passes through nodes $e_1, e_2, \ldots, e_s$. The path is said to admit a shortcut if there exist an arc (e_k, e_j) in BG(I), where $1 \leqslant k \leqslant j-2 \leqslant s-2$.

Lemma 4.1.8 : If S is a source-sink path in BG(I) which admits no shortcut, then S is an augmenting sequence with respect to I.

Lemma 4.1.9 : Let I, J be intersections such that $|I|+1=|J|$. There exists a source-sink path S in BG(I) where $S \subseteq I \oplus J$.

The above ideas may become clearer with the help of an example:

Example: Let G_1 and G_2 be the graphs shown in figure 1. Each graph is constructed from the set of arcs, which, for notational convenience, we denote E={1,2,3,4,5,6,7}. Let M_1=(E,$\mathbf{I}_1$) and M_2=(E,$\mathbf{I}_2$) be the graphic matroids associated with their respective graphs. I={2,4,6} is an independent set in both matroids. Note that I is a maximal set in $\mathbf{I}_1 \cap \mathbf{I}_2$ but it is not a maximum.

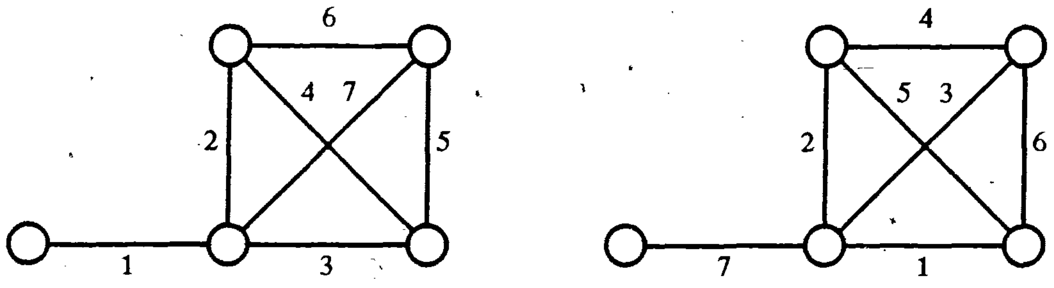


Figure 1. The graphs G_1 (left) and G_2 (right)

$sp_1(I)$	=	$\{2,3,4,5,6,7\}$	$sp_2(I)$	=	$\{1,2,3,4,5,6\}$
$C_3^{(1)}$	=	$\{2,3,4\}$	$C_1^{(2)}$	=	$\{1,2,4,6\}$
$C_5^{(1)}$	=	$\{4,5,6\}$	$C_3^{(2)}$	=	$\{2,3,4\}$
$C_7^{(1)}$	=	$\{2,6,7\}$	$C_5^{(2)}$	=	$\{4,5,6\}$

From this information we can create BG(I). To avoid cluttering the diagram we represent a pair of arcs of the form (x,y) and (y,x) by a single line with no arrowheads.

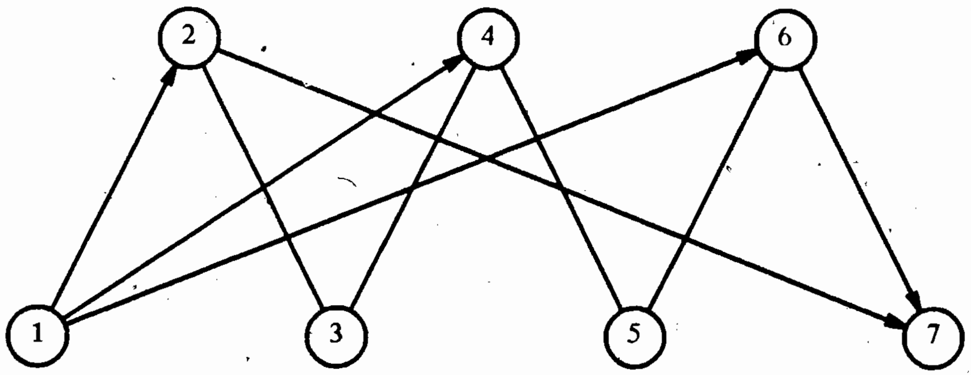


Figure 2. Border Graph for I={2,4,6}

Note that our original matroid elements (the arcs in the graphs G_1 and G_2) are now considered as vertices in the border graph. Vertex 1 is the only source; vertex 7 is the only sink.

$\{1,2,7\}$ and $\{1,6,7\}$ are two possible augmenting sequences. Although $\{1,2,3,4,5,6,7\}$ is a source to sink path in BG(I), $I \oplus \{1,2,3,4,5,6,7\} = \{1,3,5,7\}$ which is not in $\mathbf{I}_1$. In particular, $(1,2,3,4,5,6,7)$ is not an augmenting sequence. The reason the above set fails to produce an augmenting sequence is the existence of the arcs $(1,4)$, $(1,6)$ and $(2,7)$ in BG(I). Each arc allows us to take a shortcut in our path from the source to the sink.

4.1.1. Sequential Algorithm

Given two matroids $M_1=(E,\mathbf{I}_1)$ and $M_2=(E,\mathbf{I}_2)$, the following algorithm produces a set $I \in \mathbf{I}_1 \cap \mathbf{I}_2$ of maximum cardinality. In the algorithm, lines 2 to 17 are repeated for each augmentation. Each augmentation can be divided into 3 steps:

(1) Lines 3 to 5 build the border graph BG(I) where I is the intersection currently being augmented. Line 5 marks all the sources with "+". We regard the matroid elements as vertices in BG(I).

(2) Lines 6 to 11 perform a breadth-first search of BG(I). Line 9 checks if the element chosen is a sink. If it is a sink, then a source-sink path is found.

(3) Lines 12 to 16 perform the augmentation by backtracking through the source-sink path. If no source-sink path is found, then the current intersection is of maximum cardinality, and the algorithm will stop.

Algorithm 4.1 : Sequential Cardinality Algorithm [La-76]

```
1.      $I \leftarrow \emptyset$;  Q $\leftarrow \emptyset$;
2.      while not done
3.         for each $e_i$ in $E-I$
4.            Find $C_i^{(1)}$ and $C_i^{(2)}$ if they exist.
5.            Add each vertex $e_i \in E-sp_1(I)$ to Q with "+0" mark
6.         while Q$\neq\emptyset$ and augmenting sequence not found
7.            Remove first element $e_i$ from Q
8.            if "+" mark then
9.               if $I \cup \{e_i\} \in \mathbf{I}_2$ then augmenting sequence found
10.              else add each unmarked $e_j \in C_i^{(2)}$ to Q with "-i" mark
11.           else add each unmarked $e_j$ such that $e_i \in C_j^{(1)}$ to Q with "+i" mark
12.        if augmenting sequence found then
13.           backtrack from $e_i$ (found in line 9) to get augmenting sequence
14.           add elements of sequence with "+" marks to $I$
15.           remove elements of sequence with "-" marks from $I$
16.           Q $\leftarrow \emptyset$ and remove marks from all elements
17.        else done
```

The following algorithm finds the circuit $C_i^{(j)}$ in M_j if it exists.

Find_Circuits

Find the unique circuit $C_i^{(j)}$ in matroid M_j contained in the set $I \cup \{e_i\}$ if such a circuit exists.

```
1.   $C_i^{(j)} \leftarrow \emptyset$
2.   if $I \cup \{e_i\} \in \mathbf{I}_j$ then no circuit
3.   else
4.       for each $e_k \in I$
5.           if $I \cup \{e_i\} - \{e_k\} \in \mathbf{I}_j$ then $C_i^{(j)} \leftarrow C_i^{(j)} \cup \{e_k\}$
```

Time Analysis

Let $c_1(m)$ and $c_2(m)$ be the running times of the subroutines for independence testing in M_1 and M_2 respectively where $m = |E|$.

Let $c(m) = \max\{ c_1(m), c_2(m) \}$

Let R_1 and R_2 be the ranks of the matroids M_1 and M_2 respectively, and let $R = \min\{ R_1, R_2 \}$

Since no intersection can contain more than R elements, there can be no more than R augmentations in the algorithm. For each augmentation, we have the following steps:

(1) For each $e_i \in E - I$, we find the circuits $C_i^{(1)}, C_i^{(2)}$. This can be done by testing the independence of $I + e_i - e_j$ for each $e_j \in I$. There will be $O(R)$ elements in I and $O(m)$ elements in $E - I$. Therefore the time taken will be $O(mRc(m))$.

(2) During marking, there will be $O(R)$ elements with a "-" mark. For each such element e_i we check every element $e_j \in sp_1(I) - I$ to see if it forms an M_1-circuit that contains e_i. There will be $O(m)$ elements in $sp_1(I) - I$. Hence there are $O(mR)$ checks for "-" marked elements. Similarly, there will be $O(m)$ elements with a "+" mark. For each of such element e_i, we check every element e_j in $C_i^{(2)}$. There are $O(R)$ elements in $C_i^{(2)}$. Hence there are $O(mR)$ checks for "+" marked elements. We conclude that the marking takes $O(mR)$ time.

(3) Backtracking takes $O(R)$ time.

Therefore, the overall running time is $O(mR^2c(m))$.

Note that in [La-76], the time complexity given for this algorithm is $O(m^2R + mR^2c(m))$ because it assumes that marking takes $O(m^2)$ time. We have improved on this analysis in the above.

4.1.2. Parallel Algorithm

In this section, we shall present a parallel algorithm to solve the cardinality two-matroid intersection problem. The algorithm is derived from the sequential algorithm. For the time being, we assume that the algorithm uses m^2 processors. In Section 4.4.3 we shall show how to reduce the number of processors without increasing the time bound so that perfect speed-up can be achieved.

In the parallel algorithm, m^2 processors, P_{ij}, $1 \leqslant i, j \leqslant m$ will be used, one for each possible link in BG(I), the border graph for the current intersection I. The following variables are used.

$LINK_{ij}$	: (local variable) true if $(e_i, e_j) \in BG(I)$ for $1 \leqslant i, j \leqslant m$.

The following are global variables for $1 \leqslant j \leqslant m$.

I	: boolean vector indicating membership of elements in the current intersection. $I(i)$ is true iff e_i is in the current intersection.
$SOURCE_j$	: true if e_j is a source in $BG(I)$
$SINK_j$	: true if e_j is a sink in BG (I)
$ACTIVE_j$	: true if e_j is "active"
$PARENT_j$	: i if e_i is the parent of e_j in forest "F" (it is null initially)
AUG	: current node on source to sink path during backtracking (backtracking starts at a sink)
$ENDSEARCH$	: true if the search for source-sink path is done

The terms "active" and "F" will be explained in the following.

Steps 1 to 4 of the algorithm are executed once for each augmentation. Step 1 is to initialize some variables and takes constant time. Step 2 builds BG(I). Processor P_{ij} will set $LINK_{ij}$ true if and only if it finds that an arc from e_i to e_j exists in BG(I). $SOURCE_i$ ($SINK_i$) is set true if and only if e_i is a source (sink) in BG(I). The proof of correctness in a later section will show how this can be done in O($c(m)$) time.

Step 3 will find a source-sink path with no shortcut in BG(I) if it exists. This is done by growing directed trees with the sources as the roots and the links in BG(I) as the branches. We shall call the forest of these trees "F". We will see that each element has at most one parent in F while it may have many parents in BG(I). Since the border graph is traversed in a breadth-first manner, the depth of this search is the length of the shortest source-sink path and this is no longer than $2R$. Each level of this search is done in parallel in constant time, so that a time complexity of O(R) results.

The while loop in lines 4-8 of Step 3 will be iterated once for each level of the search. The sources are the "active" nodes for the first iteration of the while loop. In each iteration, each active node, e_i, will examine all its sons. If a son e_j has no assigned parent ($PARENT_j$ = null), then, e_i will attempt to make itself the parent of e_j in F. There may be more than one active node attempting to be the parent of the same node, e_j. In our parallel model, one processor will succeed and it is not important which one succeeds because any search path to e_j at this point will have the same length. If e_j is a sink, then a source-sink path is found. AUG is set to j, and Step 3 is finished. If e_j is not a sink, then it is activated at line 5. Except for the first iteration, a node will be "active" if and only if it has been assigned a parent in the previous iteration. All nodes e_i activated in the previous iteration will be deactivated at line 8. Hence a node will never be assigned a parent in F for a second time, and it will be active for at most one iteration.

If no source-sink path exists, then the search will reach a point at which all nodes have been activated. Since no node will be activated at the next iteration, ENDSEARCH will become true and the search stops.

Step 4 performs the augmentation of I. At line 1, if AUG equals null, then no source-sink path was found in Step 3. This implies that the current I is of maximum cardinality and the algorithm halts. Otherwise, I is augmented by backtracking through the source-sink path starting from the sink. AUG will be the current node in the path during the backtracking. Since the path is at most $2R$ long, we do not lose any time efficiency by backtracking element by element.

In the following algorithm, the sign '$\leftarrow^*$' means concurrent write, the sign '$\leftarrow^{**}$' means concurrent write of possibly different values to the same location.

Algorithm 4.2 : Parallel Cardinality Algorithm

Input : Two matroids $M_1=(E,\mathbf{I}_1)$ and $M_2=(E,\mathbf{I}_2)$.
Output : An intersection I in $\mathbf{I}_1 \cap \mathbf{I}_2$ of maximum cardinality.

```
    Each $P_{ij}$ ; $1 \leq i, j \leq m$ executes the following code.
    Let $I$ be in the intersection of $M_1 = (E, \mathbf{I}_1)$ and $M_2 = (E, \mathbf{I}_2)$.
    ($I$ can be $\emptyset$)

Step 1 : (Initialization)
1.      $LINK_{ij} \leftarrow$ false ;  $ACTIVE_j \leftarrow^*$ false ;  $PARENT_j \leftarrow^*$ null ;   $AUG \leftarrow^*$ null
2.      if $e_j \notin I$ then
            $SOURCE_j \leftarrow^*$ true ;   $SINK_j \leftarrow^*$ true
        else
            $SOURCE_j \leftarrow^*$ false ;  $SINK_j \leftarrow^*$ false
        <synchronize>

Step 2 : (Build border graph $BG(I)$)
1.      if $e_i \in I$ and $I + e_j \notin \mathbf{I}_1$ and $I + e_j - e_i \in \mathbf{I}_1$ then
            $LINK_{ij} \leftarrow$ true ;  $SOURCE_j \leftarrow^*$ false
2.      if $e_j \in I$ and $I + e_i \notin \mathbf{I}_2$ and $I + e_i - e_j \in \mathbf{I}_2$ then
            $LINK_{ij} \leftarrow$ true ;  $SINK_i \leftarrow^*$ false
        <synchronize>
```

```
Step 3 : (Find source to sink path)
1.        if $SOURCE_j$ and $SINK_j$ then
            $I \leftarrow^* I + e_j$     (path is found)
          <synchronize>
          if path is found then go to Step 1
2.        $ENDSEARCH \leftarrow^*$ true.
3.        if $SOURCE_j$ then
            $PARENT_j \leftarrow^* 0$ ;   $ACTIVE_j \leftarrow^*$ true ;   $ENDSEARCH \leftarrow^*$ false
4.        while path not found and not $ENDSEARCH$
          <synchronize>
            $ENDSEARCH \leftarrow^*$ true
5.          if $LINK_{ij}$ and $ACTIVE_i$ and $PARENT_j$ = null then
              $ACTIVE_i \leftarrow^*$ false
6.            $PARENT_j \leftarrow^{**} i$
7.            if $SINK_j$ then
                $AUG \leftarrow^{**} j$  (path found)
              else
8.              $ACTIVE_j \leftarrow^*$ true ;   $ENDSEARCH \leftarrow^*$ false
            <synchronize>
          end-while

Step 4 : (Augmentation)
1.        if $AUG$ = null then stop  ($I$ has maximum cardinality)
2.        while augmentation not done
3.          if $j = AUG$ and $i = PARENT_j$ then
4.            if $SOURCE_i$ then
                $I \leftarrow^* I + e_i$     (augmentation is done)
              else
5.              if $e_j \in I$ then $I \leftarrow^* I - e_j$  else $I \leftarrow^* I + e_j$
                $AUG \leftarrow^* i$
          <synchronize>
          end-while
          go to Step 1
```

Proof of Correctness

If we can prove that each iteration of the algorithm

(1) builds the border graph BG(I) for the current I and

(2) discovers a source-sink path with no shortcut in BG(I) whenever one exists,

then by Theorems 4.1.2, 4.1.3, 4.1.4, 4.1.5 and Lemmas 4.1.8, 4.1.9, the algorithm is correct.

(1) Line 1 in Step 2 checks the following for each $e_i \in I$:

If $I+e_j \notin \mathbf{I}_1$ and $I+e_j-e_i \in \mathbf{I}_1$, then e_i is in the unique circuit $C_j^{(1)}$ in $I+e_j$, and the link from e_i to e_j is established. This corresponds exactly to the first half of BG(I) (part (i) of Definition 4.1.6). Similarly, line 2 in Step 2 constructs the second half of BG(I) (part (ii) of Definition 4.1.6).

We must also make sure that the sources and sinks are correctly marked. During initialization, if $e_i \notin I$ it is both a sink and a source; otherwise both $SOURCE_i$ and $SINK_i$ are initialized to false. In line 1 of Step 2, for any e_j, e_j is marked not to be a source if and only if $e_j \in sp_1(I)$ and $e_j \notin I$. Therefore, e_j is marked to be a source if and only if $e_j \notin sp_1(I)$. In line 2 of Step 2, for any e_i, e_i is marked not to be a sink if and only if $e_i \in sp_2(I)$ and $e_i \notin I$. Therefore e_i is marked to be a sink if and only if $e_i \notin sp_2(I)$.

(2) The sequential algorithm performs a breadth-first search of the border graph from the sources. This search will uncover a source-sink path with no shortcut whenever one exists. In fact, such a search will give a shortest source-sink path.

In Step 3, the algorithm starts from the sources and examines the descendents level by level in parallel. The first time that an element is examined, it is marked by assigning a *PARENT* number. An element will not be marked in more than one of the while loops.

Define "possible trees" to be the fully generated directed trees rooted from all the sources of BG(I). A breadth-first search will be incorrect in this algorithm only if some node for an element at a higher level (further from a source) of the possible trees is inserted while a node for the same element at a lower level (i.e. closer to a source) is not. The possible trees are examined level by level by the while loop which begins at line 4. Since the while loop is synchronized, nodes at a higher level will not be examined until all nodes at lower levels are searched. Hence the above violation of breadth-first order will never occur and Step 3 is a breadth-first-search. Since the search at each level is done in parallel, the outcome will be a randomly ordered search. In other words, a node may be marked simultaneously in the same iteration of the while loop by more than one parent, and the parallel model that we use allows such concurrent writes with the result that one of them succeeds. This causes no problems since, in path retrieval, any one of these choices will lead to a source-sink path of the same length. ☐

Time Analysis

The symbols m, $c(m)$, and R, are the same as in the time analysis of Algorithm 4.1.

(1) Steps 0 and 1 take constant time.

(2) Step 2 requires $O(c(m))$ time to build BG(I).

(3) Step 3 requires at most $2R$ iterations of the while loop.

Hence it takes $O(R)$ time.

(4) Step 4 requires $O(R)$ time for backtracking.

There are at most R iterations of steps 1 to 4.

Therefore the overall running time is $O(R(R+c(m)))$.

Remarks

It will be shown in section 4.4.3 that Step 4 can be speeded up by *recursive doubling* to $O(\log R)$ time. However, it is more difficult to reduce Step 3 to the same efficiency. The algorithm has a strong sequential nature because each iteration of the while loop depends on the previous iterations when it examines PARENT() to check if a node has been previously marked. We shall see in a later stage how to do Step 3 in $O(\log R)$ time but with more processors.

The $O(R)$ factor due to the $O(R)$ augmentations is also difficult to reduce because each augmentation depends on the previous augmentations. It may require an entirely different strategy than the existing sequential algorithms. We do not expect this to be easy to find.

4.2. WEIGHTED TWO-MATROID INTERSECTION PROBLEM

The second problem to be considered in this chapter is the weighted two-matroid intersection problem. Here, the elements in the two matroids are weighted and we have to find a maximum weight set independent in both matroids.

The method for solving this problem is similar to the unweighted problem in that we also proceed by computing intersections of successively greater sizes. These intersections also increase in size by one each time. However, during each search for a larger intersection, we must also make sure that the new intersection has maximum weight among all intersections of the same size.

The algorithms in this section are generalizations of the unweighted intersection algorithms of the previous section which were based on augmenting sequences and border graphs. However, this time we do not stop at an arbitrary source-sink path with no shortcut because it may not give a new set with maximum weight among all possible source-sink paths. Instead, an exhaustive search for all source-sink paths is made. To determine the weight of each new intersection, we will record *incremental weights* during the search. Suppose at a certain point of the search, we have a temporary set $I+e_1-e_2+e_3-\ldots$.. Then the incremental weight of the alternating sequence $\{e_1,e_2,e_3 \cdots\}$ is

$$\mathrm{wt}(e_1) - \mathrm{wt}(e_2) + \mathrm{wt}(e_3) - \ldots$$

where $\mathrm{wt}(e_i)$ is the given weight for element e_i.

Hence, if a new intersection is $I+e_1-e_2+e_3- \cdots +e_s$, then the weight of this set is

$$(\text{weight of I}) + \mathrm{wt}(e_1) - \mathrm{wt}(e_2) + \mathrm{wt}(e_3) - \ldots + \mathrm{wt}(e_s)$$

which is equal to the sum of weight of I and the incremental weight of the augmenting sequence $\{e_1,e_2,e_3, \cdots e_s\}$. To find a new intersection of maximum weight, we have to find an augmenting sequence of maximum incremental weight.

As stated below (Theorem 4.2.4), the incremental weights for consecutive augmentations are non-increasing. Hence, if we find an augmentation that gives a non-positive incremental weight, we know that the present intersection has maximum weight and the problem is solved. The following definitions and theorems are from [La-76].

Definition 4.2.1 : Given an intersection I, and a set $S \subseteq E$, the *incremental weight* of S is

$\Delta(S)$ = weight of $\{S - I\}$ - weight of $\{S \cap I\}$

Clearly, weight of $\{I \oplus S\}$ = weight of $\{I\}$ + $\Delta(S)$

Definition 4.2.2 : An intersection I is *p-maximal* if $|I| = p$ and I is of maximum weight among intersections containing p elements.

Theorem 4.2.3 : Let I be a *p-maximal* intersection and S be a maximum incremental weight source-sink path in BG(I). Then S is a *maximum weight augmenting sequence* and $I \oplus S$ is *(p+1)-maximal*.

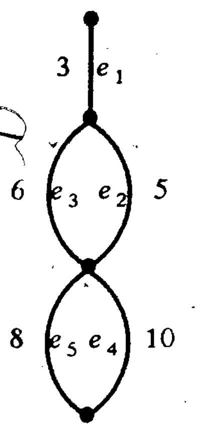
Theorem 4.2.4 : Let I_{p-1}, I_p, and I_{p+1} be intersections which are $(p-1)$-, p-, and $(p+1)$- maximal respectively. Then

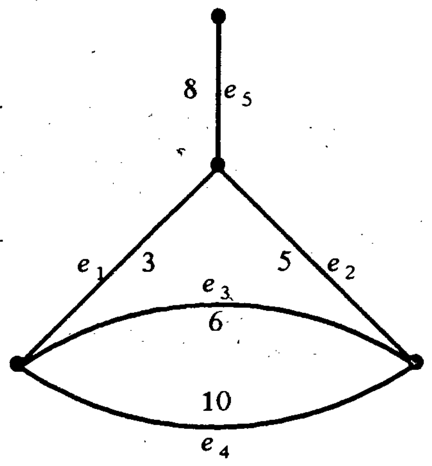
$$w(I_p) - w(I_{p-1}) \geqslant w(I_{p+1}) - w(I_p).$$

where $w(I_i)$ is the weight of I_i.

An example will help to illustrate these ideas.

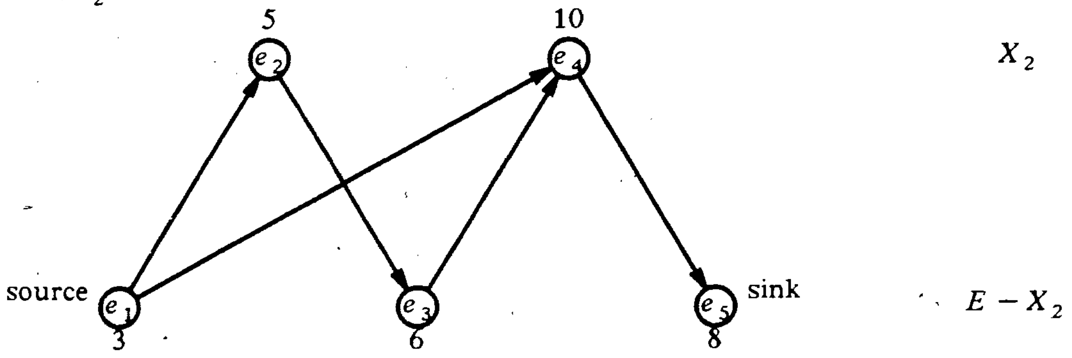
Example. $X_2 = \{e_2, e_4\}$ is a 2-maximal intersection for the graphic matroids M_1 and M_2 below.





M_1 M_2^1

$BG(X_2)$ is :



There are two augmenting sequences $S_1 = \{e_1, e_4, e_5\}$ and $S_2 = \{e_1, e_2, e_3, e_4, e_5\}$. $X_2 \oplus S_1 = \{e_1, e_2, e_5\}$ with $\Delta(S_1) = 1$ and $X_2 \oplus S_2 = \{e_1, e_3, e_5\}$ with $\Delta(S_2) = 2$. Thus the maximum weight augmenting sequence is S_2 and $X_3 = \{e_1, e_3, e_5\}$.

4.2.1. Sequential Algorithm

The following sequential algorithm from [La-76] solves the weighted two-matroid intersection problem using the above method. Lines 3 to 28 find an augmentation and they are repeated until the problem is solved. Each augmentation consists of three parts:

(1) Lines 5 to 9 build the border graph BG(I) for the current intersection I. The sources are marked with "+".

(2) Lines 10 to 25 perform a breadth-first search of BG(I) to obtain a maximum incremental weight source-sink path. Line 13 checks for a sink.

(3) Lines 26 and 27 perform augmentation of I by backtracking through the source-sink path

found in part 2. If the incremental weight is non-positive, then the current intersection has maximum weight and the algorithm will stop.

Algorithm 4.3 : Sequential Weighted Algorithm [La-76]

```
1.  I ← $\phi$
2.  queue is empty
3.  while not done
4.      $\Delta(S) \leftarrow -\infty$
5.      For each $e_i \in E - I$
6.          Find $C_i^{(1)}$ and $C_i^{(2)}$ if they exist
7.          $\Delta(e_i) \leftarrow -\infty$
8.          Add each source node $e_i \in E - sp_1(I)$ to queue with "+0" mark
9.          and set $\Delta(e_i) \leftarrow w_i$
10.     while queue not empty
11.         Remove first element $e_i$ from queue
12.         If "+" mark then
13.             If $I + e_i \in I_2$ then
14.                 if $\Delta(e_i) > \Delta(S)$ then
15.                     $\Delta(S) \leftarrow \Delta(e_i)$
16.                     $S \leftarrow i$
17.                 Else
18.                     Add each $e_j \in C_i^{(2)}$ where $\Delta(e_j) < \Delta(e_i) - w_j$
                            to queue with "-i" mark
19.                     $\Delta(e_j) \leftarrow \Delta(e_i) - w_j$
20.                     If another $e_j$ exists in the queue remove it from queue
            Else
21.             Add each $e_j$ such that $e_i \in C_j^{(1)}$ and $\Delta(e_i) + w_j > \Delta(e_j)$
                    to queue with "+i" mark and update
22.             $\Delta(e_j) \leftarrow \Delta(e_i) + w_j$
23.             If another $e_j$ exists in the queue remove it from queue
    end-while
24. If $\Delta(S) > 0$ then
        Backtrack from $e_s$ to get augmenting sequence
        Add elements with "+" marks to I
        Delete elements with "-" marks from I
25.     Remove all marks and empty queue
    Else
26.     Done ( if $\Delta(S) = -\infty$ then I has maximum cardinality)
```

Time analysis

(1) It takes $O(mRc(m))$ time to compute $C_i^{(1)}$ and $C_i^{(2)}$ for all $e_i \in E - I$.

(2) Each of the $O(m)$ elements in $E-I$ may receive $O(R)$ marks from its parents in BG(I) and each marking requires $O(R)$ time. Each of the $O(R)$ elements in I may receive $O(R)$ marks

and each marking requires $O(m)$ time. Therefore the marking takes $O(mR^2)$ time.

(3) Backtracking requires $O(R)$ time.

There can be at most R augmentations. Hence the overall running time is $O(mR^3+mR^2c(m))$

In [La-76], the time complexity given for this algorithm is $O(m^2R^2+mR^2c(m))$ because it claims that the labeling procedure consumes $O(m^2R)$ time per augmentation. In the above, we have improved on this analysis.

4.2.2. Parallel Algorithm

In this section, we present a parallel algorithm for the weighted two-matroid intersection problem which has been derived from the sequential algorithm above. Again, we assume that m^2 processors are used for the time being and show how to reduce this number in a later section to achieve perfect speed-up

The parallel algorithm for the weighted matroid intersection problem is similar to the parallel algorithm for the cardinality problem. We use m^2 processors P_{ij} where $i,j = 1,2,...m$. (m is the number of elements in the matroids.) Each processor handles a possible link in the border graph. The following variables are used.

The following are local variables:

WEIGHT(i) : given weight for element e_i.

WT_i : the weight contributed by element e_i to the incremental weight of a source-sink path through e_i.

$LINK_{ij}$: (local variable) true if $(e_i, e_j) \in BG(I)$.

The following are global variables.

$SOURCE_j$: true if e_j is a source in BG(I).

$SINK_j$: true if e_j is a sink in BG(I).

$ACTIVE_j$: true if e_j is "active".

$PARENT_j$: i if e_i is parent of e_j in "F".

Δ_i : greatest incremental weight among all paths from a source to e_i.

Δ_s : weight of the augmenting sequence.

$ENDSEARCH$: true if the search for source-sink path is done.

AUG : current node on source-sink path during backtracking.

The terms "active" and "F" are explained in the following.

Steps 1 to 4 of the algorithm are executed once for each augmentation. Step 1 initializes variables and takes constant time. Step 2 builds BG(I) in the same way as in Algorithm 4.2. Step 3 will find a source-sink path with the greatest incremental weight if it exists. This path will correspond to a maximum weight augmenting sequence. Step 3 is a breadth-first search of BG(I) starting from the sources. The search will build a forest, F, of trees with the sources as the roots and links in BG(I) as the branches. Each element has at most one parent in F while it can have many parents in BG(I). The depth of this search is the length of the maximum weight augmenting sequence and this is no longer than $2R$. Each level of this search is done in parallel and a time complexity of O(R) results.

The while loop in lines 3-8 of Step 3 will be iterated once for each level of the search. Before the first iteration, each element e_j is assigned a local weight WT_j. WT_j is the given weight $WEIGHT(j)$ if $e_j \notin I$ and $-WEIGHT(j)$ if $e_j \in I$. All sources are assigned a parent of 0 and they are the "active" elements in the first iteration of the while loop.

For each element e_i, a variable Δ_i will record the greatest incremental weight among all paths from the sources to e_i that have been searched so far. At line 4 in the while loop, each active node e_i will generate all its sons in BG(I). Each son, e_j, will check whether $\Delta_i + WT_j$ is greater than the current Δj. There can be more than one active parent e_i generating e_j and the maximum $\Delta_i + WT_j$ is recorded in Δ_j using the write-max instruction. The corresponding e_i becomes the parent of e_j in F. (i.e., $PARENT_j \leftarrow i$)

If e_j is a sink then we have a source-sink path which is a potential augmenting sequence. The write-max at line 7 will retain only the source-sink path with the maximum incremental weight. The sink of the chosen path will be entered into AUG. If e_j is not a sink then it is activated at line 8. The search will continue until all paths from the sources are exhausted. Except for the first iteration, a node will be "active" if and only if it is not a sink and it has been assigned a "new" parent in F in the previous iteration. Note that an element may be assigned a different parent during each iteration. All nodes e_i activated in the previous iteration will be deactivated at line 6. However, the node can be activated aga'n at line 8 in the same iteration. Hence, a node may be activated in more than one iteration of the search.

Step 4 performs the augmentation of I by backtracking in the same way as Algorithm 4.2. It will first check the value of Δ_s. If it is non-positive then I is a solution and the algorithm stops.

Algorithm 4.4': Parallel Weighted Algorithm

Let I be any $|I|$-maximal intersection of M_1, M_2.
(I can be $\varnothing$ initially.)
Each P_{ij}, $1 \leqslant i, j \leqslant m$ executes the following codes

Step 1 : (* *Initialization* *)

```
1.    $LINK_{ij} \leftarrow$ false;   $ACTIVE_j \leftarrow^*$ false;   $PARENT_j \leftarrow^*$ null;   $AUG \leftarrow^*$ null
      $LINK_{ii} \leftarrow$ true;   $\Delta_s \leftarrow^* -\infty$
?.    $SOURCE_j \leftarrow^*$ true;   $SINK_j \leftarrow^*$ true
      if $e_j \in I$ then
          $SOURCE_j \leftarrow^*$ false ;   $SINK_j \leftarrow^*$ false
```

Step 2 : (* *Building Border Graph BG(I)* *)

```
1.    if $e_i \in I$ and $I+e_j \notin I_1$ and $I+e_j-e_i \in I_1$ then
          $LINK_{ij} \leftarrow$ truer;   $SOURCE_j \leftarrow^*$ false
2.        if $e_j \in I$ and $I+e_i \notin I_2$ and $I+e_i-e_j \in I_2$ then
              $LINK_{ij} \leftarrow^*$ true;   $SINK_i \leftarrow^*$ false
```

Step 3 : (* *search for source-sink path* *)

```
1.    if $i \notin I$ then $WT_j \leftarrow^*$ $WEIGHT(j)$ else $WT_j \leftarrow^*$ $-WEIGHT(j)$
      <synchronize>
      $ENDSEARCH \leftarrow^*$ true
2.    if $SOURCE_j$ then
          $\Delta_j \leftarrow WT_j$;   $ACTIVE_j \leftarrow^*$ true;        $ENDSEARCH \leftarrow$ false
```

```
3.    while not $ENDSEARCH$
      <synchronize>
              $ENDSEARCH \leftarrow^{*}$ true
4.            if $ACTIVE_i$ and $LINK_{ij}$ then
5.                    write-max( $\Delta_j$ , $\Delta_i + WT_j$ )
              <synchronize>
                      $ACTIVE_i \leftarrow^{*}$ false
6.                    if $\Delta_j = \Delta_i + WT_j$ then $PARENT_j \leftarrow^{**} i$
7.                    if $SINK_j$ then
                              write-max($\Delta_s$ , $\Delta_j$ )
                      <synchronize>
                              if $\Delta_s = \Delta_j$ then $AUG \leftarrow^{**} j$

8.                    else
                              $ENDSEARCH \leftarrow^{*}$ false;  $ACTIVE_j \leftarrow^{*}$ true
              <synchronize>
      end-while
```

Step 4 : (* *augmentation* *)

```
1.    if $\Delta_s = -\infty$ then
              $I$ is both maximum cardinality and maximum weight, STOP
      else
              if $\Delta_s \leqslant 0$ then $I$ is of maximum weight, STOP

      if $SOURCE_{AUG}$ then $I \leftarrow I + e_{AUG}$
      else
2.    while augmentation not done
3.            if $i = PARENT_j$ and $j = AUG$ then
                      if $SOURCE_i$ then $I \leftarrow^{*} I + e_i$   (* augmentation is done *)
                      else
                              if $e_j \in I$ then $I \leftarrow^{*} I - e_j$ else $I \leftarrow^{*} I + e_j$
                              $AUG \leftarrow^{*} i$

      <synchronize>
      end-while
```

Proof of Correctness

Theorems 4.2.3 and 4.2.4 give us the following algorithm for the weighted matroid intersection problem:

```
Start with any $p$-maximal intersection $I$
Repeat the following until done :
    Build border graph BG($I$)
    and find a maximum incremental weight
    source-sink path $S$ in BG($I$).
    If S has positive incremental weight then
    $I \leftarrow I \oplus S$ else done .
```

We want to show that this is what the parallel algorithm does.

(1) Step 2 is the same as in the cardinality algorithm and we have proved that it builds BG(I) correctly.

(2) Next we prove that Step 3 will find a source-sink path with maximal incremental weight in BG(I). Step 3 is a breadth-first search of all trees in BG(I) with roots at the sources. The search keeps track of the temporary incremental weight Δ_i of an optimal path to each node e_i from a source. If there are several paths coming to the same node e_i, only the path with the maximum Δ_i value can lead to the final maximum incremental weight path. Therefore, we need to prove that the algorithm will retain only this path for possible backtracking. This is done in the while loop of Step 3.

A node e_i is activated if it is not a sink and it is assigned a new parent. e_i will examine each of its sons e_j to see if Δ_j would increase if e_i became the parent of e_j in F. If Δ_j can be increased, then Δ_j is updated and $PARENT_j$ becomes i. This means that e_i is parent of e_j in F, and, during backtracking, node e_j will point to e_i.

Since the search is parallel at each level, there can be several parent nodes attempting to activate node e_j simultaneously. The write-max instruction at line 5 ensures that each of these parent nodes gets a chance to activate e_i. A parent in BG(I) giving the maximum Δ_j

value will succeed and become the parent of e_j in F.

Similarly, there can be more than one source-sink path with an incremental weight greater than the current value of Δ_s. The write-max instruction at line 7 ensures that the path with maximum Δ_s value becomes the augmenting sequence.

(3) By Theorem 4.2.4, the incremental weights for consecutive augmentations in this algorithm are decreasing. Therefore, when we reach an augmenting sequence with non-positive incremental weight, we know that all the following augmentations will give decreasing weights, so the existing intersection is maximum weight. Hence, Step 4 correctly stops the computation when it discovers a non-positive Δ_s. □

Time Analysis :

(1) Steps 0 and 1 take constant time.

(2) Step 2 takes $O(c(m))$ time to build the border graph $BG(I)$.

(3) In Step 3, the while loop will be repeated $O(R)$ times. Each iteration takes constant time, so the total time for Step 3 is $O(R)$.

(4) Backtracking in Step 4 takes $O(R)$ time.

There will be at most R iterations of steps 1 to 4. Therefore, the overall running time of the algorithm is $O(R(R+c(m)))$.

4.3. REDUCING THE NUMBER OF PROCESSORS

The parallel algorithms for solving the two-matroid intersection problem described in the previous two sections require m^2 processors. It can be seen that much of the time, many of the processors are idle. For example, when building the border graph, only those processors P_{ij} for which ($e_i \in E-I$ and $e_j \in I$) or ($e_i \in I$ and $e_j \in E-I$) can do useful work. This is because $E-I$ and I are the two vertex sets in the bipartite graph BG(I). Also, when searching for a source-sink path in BG(I), only processors P_{ij} for which e_i is active at this instant and e_j is a son of e_i in BG(I) will be used. If $e_i \in I$ then there can be at most O(R) such active elements. If $e_i \in E-I$ then there can be at most O(R) sons for e_i. In both cases, there will be $O(m(m-R))$ processors idle. The existence of idle processors is also found in the other steps.

With all this waste in processing power, we cannot expect perfect speed-up. Fortunately this can be improved by using Brent's Theorem [SV1-82]. This theorem says that if the total number of elementary operations (operations that takes O(1) time using one processor) that all the processors together will perform is x, and if the parallel time (depth) required is d, then we can implement the algorithm with x/d processors with the same depth d if we know how many elementary operations there will be at each instant and we know how to distribute them to the x/d processors. So, if the number of operations x is the same as for the sequential algorithm, then the time-processor product will be equal to x, the sequential time, and a perfect speed-up will result. Since the parallel matroid intersection algorithms are doing the same things as the sequential algorithms, we would expect that the number of operations are the same too. In fact, we find that this is true, and a perfect speed-up is possible for these algorithms. A similar application of Brent's Theorem can be found in [SV1-82] in which a parallel max-flow algorithm is designed. Brent's Theorem is stated as follows:

Theorem 4.3.1 : (Brent) Any synchronized parallel algorithm of depth d that consists of a total of x elementary operations can be implemented by p processors within a depth of $\left\lceil \frac{x}{p} \right\rceil + d$. (Elementary operations take O(1) time).

Our algorithms can be synchronized at each instruction so that Brent's Theorem can be applied. To apply this theorem, we need to solve two implementation problems :

(1) Determine the number of operations to be performed at each time instant.

(2) Assign the processors to their jobs.

These problems will be solved in section 4.3.3. Let us assume for now that they can be solved. To determine the optimal number of processors p, we must first determine x, the number of elementary operations in the algorithm. The analysis for the unweighted and weighted intersection algorithms are given in the next two sections.

4.3.1. The Cardinality Algorithm

Let us consider the number of elementary operations in each step of the parallel algorithm.

(1) In Step 1, the initialization needs $O(mR)$ elementary operations since this is the number of possible links in the border graph to be initialized.

(2) Step 2 builds the border graph by determining the links in it. It will perform independence tests for the $O(m)$ elements in $E-I$. Each element e_i is tested $O(R)$ times for each possible link of e_i to I. Each test requires $O(c(m))$ time which we can take to be $O(c(m))$ elementary operations. Therefore there will be $O(mRc(m))$ elementary operations in Step 2.

(3) Step 3 computes a shortest source-sink path. Each of the m elements may be activated at most once. There are $O(m)$ elements in the set $E-I$, and each of these elements will examine $O(R)$ sons when it is active. There are $O(R)$ elements in the set I, and each of these elements will reference $O(m)$ sons when it is active. Therefore the total number of elementary operations in Step 3 is $O(mR)$.

(4) Step 4 is backtracking which requires $O(R)$ operations.

Since steps 2, 3 and 4 will be repeated $O(R)$ times, (there are $O(R)$ augmentations), the total number of elementary operations for the entire algorithm is $O(mR^2c(m))$.

The depth d, or time requirement of the algorithm, has been found to be $O(R^2 + Rc(m))$.

Applying Brent's Theorem gives the following result:

Theorem 4.3.2 : Algorithm 4.2 can be implemented using p processors within a depth of

$$\left\lceil \frac{O(mR^2c(m))}{p} \right\rceil + O(R(R+c(m))).$$

The minimum number of processors to maintain the previous depth of $O(R(R+c(m))$ is $\min(mc(m), Rm)$. This gives a time-processor product of $O(mR^2c(m))$ which is the same as the time of the sequential algorithm, so a perfect speed-up is achieved.

4.3.2. The Weighted Algorithm

Let us consider the number of elementary operations in each step of Algorithm 4.4. Steps 0, 1, 2 and 4 are the same as in the cardinality algorithm. The only difference is in Step 3. Each of the m elements may be activated at most once in each of the $O(R)$ iterations of the while loop at line 3 to 8. When an element is activated, it will examine each of its sons in the border graph. There will be $O(R)$ elements in the set I and each of these elements will have $O(m)$ sons. There will be $O(m)$ elements in the set $E-I$, each of which has $O(R)$ sons. Therefore the total number of elementary operation for Step 3 is $O(mR^2)$. The number of elementary operations for each step is as follows:

Steps 0 and 1	$O(mR)$
Step 2............	$O(mRc(m))$
Step 3............	$O(mR^2)$
Step 4............	$O(R)$

Since steps 2, 3 and 4 will be repeated $O(R)$ times, the total number of elementary operations in this algorithm is $O(mR^2c(m) + mR^3)$.

The time complexity, or depth, d, of the algorithm is $O(R^2 + Rc(m))$. Hence we ~~have~~ the following results:

$$x = O(mR^2c(m) + mR^3)$$

$$d = O(R^2 + Rc(m))$$

Applying Brent's Theorem in this case, we must choose $p \geqslant mR$ to retain the depth d. These results are summarized in the following theorem:

Theorem 4.3.3 : Algorithm 4.4 can be implemented using mR processors within a depth of $O(R^2+Rc(m))$. The resulting time-processor product is $O(mR^2c(m) + mR^3)$ which is equal to the time complexity of the sequential algorithm. Therefore, we have a perfect speed up.

4.3.3. Processor Assignment

In the above discussion, when we apply Brent's Theorem, we have assumed that there is some way to assign the processors to their jobs in constant time. The method is shown below. We need the following variables to keep track of the job indices.

Icount	: the number of elements in the current independent set I
Iset [1..*Icount*]	: *Iset* [i] will be the i-th element in I
index of e_i	: reverse pointer of e_i to *Iset*
$soncount_i$	: the number of sons of e_i in BG(I)
son_i [1..$soncount_i$]	: the sons of e_i in BG(I)
actcount	: the number of active nodes at the current level of the breadth-first search for source-sink path
act [1..*actcount*]	: *act* [i] is the i-th active node

The following extra work is done for job assignment.

In Step 4, when we augment I, we do the following

```
    add one to Icount
    $k \leftarrow Icount$
    If $e_j$ is added to $I$, then also set $Iset[k] \leftarrow e_j$ and (index of $ej$) $\leftarrow k$
    If $e_j$ is deleted from $I$, then also set $k \leftarrow$ (index of $e_j$)
```

So, $Iset[1..Icount]$ will contain all elements in the augmented I and (index of $e_i \in I$) will be the index of e_i in the array $Iset$.

This ensures that in Step 2, when building BG, we know that $2.m.Icount$ jobs are to be done (the jobs for all possible links $i\,j$)

where

$i = 1,2,...m$ and

$j = Iset[1], Iset[2], \cdots Iset[Icount]$

or

$i = Iset[1], Iset[2], \cdots Iset[Icount]$ and

$j = 1,2,...m$

The k-th job will then be responsible for link $\{i, j\}$ where

```
    **if** $k \leqslant m(Icount)$ **then**
        $i = \left|\frac{k}{m}\right|$ ; $a = k \bmod m$ ; $j = Iset[a]$
    **else**
        $a = k - m(Icount)$
        $b = \left|\frac{a}{Icount}\right|$
        $i = Iset[b]$ ; $j = a \bmod Icount$
```

Assigning the p processors to these jobs is now straight forward.

In Step 2, when building BG(I), we do the following bookkeeping:

```
$soncount_i \leftarrow 1$
if $LINK_{ij}$ then
    $k \leftarrow$ fetch-and-add( $soncount_i$ ,1 )
    $son_i[k] \leftarrow j$
$soncount_i \leftarrow soncount_i - 1$
```

Hence $son_i[1..soncount_i]$ will contain all sons of e_i in BG(I). This will help us to distribute the jobs in Step 3. Let us consider the cardinality algorithm. In Step 3, in the while loop, all active elements will generate their sons. The following extra work is done at the very beginning of the loop to facilitate job assignment :

```
$actcount \leftarrow 1$
if $ACTIVE_i$ then
    $k \leftarrow$ fetch-and-add( $actcount$ ,1 )
    $act[k] \leftarrow i$
$actcount \leftarrow actcount - 1$
```

So, $act[1..actcount]$ will contain all the active elements for the search at this level. Therefore, the while loop should examine the following links ij which link the active elements to their sons:

$i = act[k]$ for $k = 1,2,\ldots actcount$

$j = son_{k[q]}$ for $q = 1,2,\ldots soncount_k$

Since each element can be activated at most once, and there are at most m elements (in $E-I$) each having at most $Icount$ sons (in I), and at most $Icount$ elements (in I) having at most m sons (in $E-I$), the maximum number of jobs in Step 3 for this iteration is $m.Icount$.

Next we apply Brent's Theorem on Step 3 alone. We shall show how job assignment can be done and determine the number of processors needed to maintain the depth of this step. Note that at each level of the breadth-first search of BG(I), either all nodes are in I, or all nodes are in $E-I$, and these two types of level alternate. Suppose we divide the $m.Icount$ jobs in the following way. At the levels where the active elements are in I, we assign m jobs for each of these O($Icount$) active elements. The k-th job is thus assigned for link ij where

$$a = \left\lfloor \frac{k}{m} \right\rfloor;\ b = k \bmod m$$

$$i = act[a];\quad j = son_i[b]$$

At the levels where the active elements are in $E-I$, we assign $Icount$ jobs for each of these O(m) active elements. The k-th job is thus assigned for link ij where

$$a = \left\lfloor \frac{k}{Icount} \right\rfloor;\ b = k \bmod Icount$$

$$i = act[a];\quad j = son_i[b]$$

Recall that the depth of Step 3 is O(R), and O($Icount$) equals O(R). If we apply Brent's Theorem for Step 3, we have O(mR) operations with depth O(R). Therefore we can use p processors within a depth of $\left\lceil \frac{O(mR)}{p} \right\rceil + O(R)$.

To maintain the depth, we must choose $p \geqslant m$. In fact we have chosen p to be min($mR, mc(m)$) (see analysis of Algorithm 4.2) and hence the job assignment problem for the whole algorithm is solved.

For the weighted problem, since each element can be activated at most R times, the arguments are similar to the above except that we shall have O(mR^2) operations with depth O(R) in Step 3. So we can choose $p \geqslant mR$. In fact, we have chosen $p = mR$ (see analysis of Algorithm 4.4) and therefore, the assignment problem is again solved.

4.4. FAST ALGORITHMS USING MORE PROCESSORS

In the previous sections, we have derived parallel algorithms for the two-matroid problems which achieve perfect speed-up. This means that we can utilize the available processor power within a constant factor. However, sometimes we want to solve a problem as fast as possible even if we need to use more processors and cannot achieve a perfect speed-up. We must still use a polynomial number of processors. It is unrealistic to speak of exponential amounts of time, processor, or memory resources. Moreover, if we use an exponential number of processors, all combinatorial problems can be solved easily by trying all possible combinations, and we do not need to design algorithms.

In this section, we shall derive faster algorithms for both the cardinality and weighted two-matroid intersection problems. Perfect speed-up is forfeited but the time complexities are reduced to almost linear time. The fast algorithms are derived from the previous parallel algorithms by improving the speed of steps 3 and 4. We can do Step 3, the search for source-sink paths, in $O(\log R)$ time using m^3 processors. Step 4, the backtracking, can be done in $O(\log R)$ time using m^2 processors. The resulting complexity for the cardinality algorithm is $O(R(\log R + c(m)))$ time using $O(m^3/\log m)$ processor. The overall complexity for the weighted algorithm becomes $O(R(\log R + c(m)))$ time using $O(m^3)$ processors.

4.4.1. Cardinality Two-Matroid Intersection

Algorithm 4.2 is designed to solve the cardinality two-matroid intersection problem. We have seen that steps 0, 1, and 2 of Algorithm 4.2 require $O(c(m))$ time. Steps 3 and 4 requires $O(R)$ time each. We shall show how to reduce the time for Step 4 in . later section. Here we show how to do Step 3 in logarithmic time.

Step 3, which is the search for a source-sink path, is a breadth-first search of the border graph $BG(I)$. This can be done in logarithmic time as follows. Initially, we build O(m) trees, one for each element e_i in E. The tree T_i for e_i will have e_i as its root (at level 0) and all the sons of e_i in $BG(I)$ at level 1. Hence, each tree will have $O(R)$ leaves. In the second stage, we examine each leaf e_j at level 1 of each tree and hook the tree with e_j as the root to this leaf. Hence, the trees now have leaves at level 2. In the third stage, we examine each leave e_k at level 2 of each tree and hook the tree T_k with e_k as the root to this leaf. In general, at the q-th stage, the trees will have a height of 2^q and leaves at the 2^q-th level are examined. This process is continued until some tree contains a source-sink path.

Note that in the above trees, each node will represent an element in the matroids (i.e., a node in the border graph BG(I)). Let us define the "*full tree*" for a tree T_i at the q-th stage to be the tree that would result if, at each of the earlier stages (stages 1 to q-1), the whole tree T_j for each leaf e_j is hooked to T_i. In a full tree, there can be many nodes representing the same element in the matroid. However, in the breadth-first search, we want at most one node at a lowest possible level (closest to the root) to be retained for each element. Hence, when we link a tree T_j to a leaf of T_i, we may not add the whole tree T_j. We check every node in T_j to see if the element it represents exists in T_i. Also, since the hooking of trees at the leaves is done in parallel, we check if some other tree is adding a node for the same element at another leaf of T_i. We do not add a node if another node is or will be in the tree T_i at a lower level (closer to the root). Hence, each element has at most one node in a tree and the size of any tree will not exceed m.

We see that the heights of the trees will be doubled at each stage. It can be shown that the maximum height of these trees is O(R), so there will be O(log$R$) stages.

Variables used in the algorithm:

$LINK_{ij}$	: (local variable) true if an arc from i to j exists in BG(I).

The followings are global variables.

$SOURCE_i$	: true if e_i is a source in BG(I).
$SINK_i$	: true if e_i is a sink in BG(I).
T_i	: tree with element e_i at the root.
$AUGT$	: the tree which contains the augmenting path.

AUG : the sink in the augmenting path.

Each element $e_j \in E$ may appear at most at one node in T_i. If e_j is in T_i then it has

(1) PARENT(i,j) = index number of its parent in T_i; and

(2) LEVEL(i,j) = the level number of e_j in T_i. (The root has level number 0.)

If e_j is not in T_i then PARENT(i,j)=null and LEVEL(i,j)=∞.

The tree T_i also has two variables:

SINK(T_i) is the index of the sink at lowest level of T_i

SINKLEVEL(T_i) is the level number of SINK(T_i)

If T_i does not contain a sink, then SINK(T_i) = null and SINKLEVEL(T_i) = ∞.

In the initialization for Step 3, we first check for possible single-element source-sink paths at line 2. If there is no such path then we build a tree T_i with root = i for each element e_i. So, element e_i is at level 0. Then we add each son, e_j of e_i to level one of T_i with *PARENT*(i,j) set to be i at line 5. If one of the sons, say e_j is a sink, then SINK(T_i) will be j, SINKLEVEL(T_i) is 1, and T_i will become inactive.

The q-th iteration of the algorithm corresponds to the q-th stage of the tree search. In the q-th iteration, for each tree T_i, each leaf e_j at level LEVEL(i,j)=2^q is examined. Each node in T_j will try to add itself to T_i. If several nodes representing the same element e_k are attempting this simultaneously, then the write-min at line 10 will choose the one at the lowest level. The parent node of e_k will be assigned accordingly. Note that the variables PARENT and LEVEL are the only variables that store the structures of the trees. Therefore, there will be no distinction between two nodes in a full tree which are at the same level having the same parent.

Now if one of the trees, T_j, added to T_i contains a sink, then SINK(T_i) will be set equal to this sink. SINKLEVEL(T_i) will become SINKLEVEL(T_j)+2^q. If more than one of the newly hooked trees contain a sink, then the sink at the lowest level, say SINK(T_k) is chosen by the write-min at line 12 to become SINK(T_i).

If the root of T_i is a source and SINK(T_i) is not null, then we have found a source-sink path. It is possible that more than one such path is found during the same iteration. The write-min at line 14 ensures that the shortest path will become the augmenting path. *AUGT* will remember the chosen tree T_i and *AUG* remembers the sink element SINK(T_i) in this tree. These will be used in Step 4 when we augment I by backtracking.

Algorithm 4.5 : Fast Cardinality Algorithm

Steps 0,1,2 are same as Algorithm 4.2.
Step 4 will be shown in section 4.4.3.

Step 3 now consists of the following :
Processor P_{ijk} does the following steps.

Initialization :

```
1.    for each element $e_i \in E$ .
2.        if $SOURCE_i$ and $SINK_i$ then $I \leftarrow^* I + \{e_i\}$  (path is found)
      <synchronize>
      if path is found then go to Step 1
3.    (* tree $T_i$ is built with root = $i$ *)
          PARENT($i,i$) ←* 0 ;  LEVEL($i,i$) ←* 0;
4.        for each element $e_j \in E$ other than $e_i$
5.            if $LINK_{ij}$ then
                  PARENT($i,j$) ←* $i$ ;  LEVEL($i,j$) ←* 1
6.                if $SINK_j$ then
                      SINK($T_i$) ←* $j$ ;      SINKLEVEL($T_i$) ←* 1
                      $T_i$ becomes inactive (* it will not grow any more *)
      <synchronize>
7.    if no element exists at level 2 ($e_i$ is a sink)
      then $T_i$ becomes inactive
      <synchronize>
```

```
q ← -1
Repeat the following until DONE:
q-th iteration :
q ← q+1
8.      for each active $T_i$
9.          for each element $e_j$ at level LEVEL($i,j$)=$2^q$
                (* $T_j$ is examined *)
10.             for each $e_k$ in $T_j$
                    write-min( LEVEL($i,k$), LEVEL($i,j$)+$2^q$ )
                <synchronize>
11.                 if LEVEL($i,k$) = LEVEL($i,j$) + $2^q$ then
                        PARENT($i,k$) ←** PARENT($j,k$)
                        (* $T_j$ is attached to $T_i$,
                          which grows twice as high *)
12.                     if SINK($T_j$) ≠ null then
                            tree $T_i$ becomes inactive
                        write-min(SINKLEVEL($T_i$), SINKLEVEL($T_j$) + $2^q$)
                        <synchronize>
13.                     if SINKLEVEL($T_i$) = SINKLEVEL($T_j$) then
                            SINK($T_i$) ←** SINK($T_j$)
14.         if SOURCE$_i$ and SINK($T_i$) ≠ null then
                DONE ←* true
                write-min( SL, SINKLEVEL($T_i$) )
            <synchronize>
15.             if SL = SINKLEVEL($T_i$) then AUGT ←* $i$
                <synchronize>
16.                 if AUGT = $i$ then AUG ←* SINK($T_i$)
17.         if no element appears at level $2^{q+1}$ then
                $T_i$ becomes inactive
            <synchronize>
```

4.4.2. Weighted Two-Matroid Intersection

Algorithm 4.4. has been designed to solve the weighted two-matroid intersection problem. Steps 0,1,2 of Algorithm 4.4 require $O(c(m))$ time. Steps 3 and 4 require $O(R)$ time each. In a later section we shall show how to speed up Step 4. Now we explain how to do Step 3 in logarithmic time.

Step 3 of the weighted algorithm, which searches for source-sink paths, is again a breadth-first search of the border graph BG(I). It can be done in logarithmic time by a method similar to Algorithm 4.5. The details are as follows.

Initially, a tree T_i is built for each element e_i in E. The tree for e_i will have e_i at its root (level 0) and the sons of e_i in BG(I) as the leaves at level 1. In the next stage, each leaf e_j at level 1 of each tree is examined and the tree T_j rooted at e_j will be hooked onto this leaf. The height of the trees becomes 2. In the third stage, each leaf e_k at level 2 of each tree is examined and the tree rooted at e_k is hooked onto this leaf. In general, at the q-th stage, the trees will have a height of 2^q and the leaves at the 2^q-th level are examined. This is repeated until the search is over.

Here we need a variable $\Delta(i, j)$ to store the greatest incremental weight among all paths from element e_i to element e_j which have been searched so far. These Δ values are attached to the nodes in the trees. In the full trees of this algorithm, there can also be many nodes representing the same element in the matroid. In the breadth-first search, we want to retain in T_i at most one node for each element e_j. This must be a node with the greatest $\Delta(i, j)$ value.

Hence, when we hook a tree T_j to a leaf of T_i, we may not add all the nodes in T_j to T_i. We must check every node in T_j to see if the element it represents exists in T_i. Also we check if some other tree is adding a node for the same element at another leaf of T_i. We do not add a node n_0 if another node n_1 representing the same element e_j is or will be in tree T_i at this stage, and n_1 has a greater $\Delta(i, j)$ value than n_0. Hence there will at most one node in T_i for each element and the size of any tree will not exceed m. The height of the trees is doubled at each stage. It can be shown that the maximum height of these trees is O(R), so there will O(log$R$) stages.

The following variables are used in the algorithm:

WEIGHT(i) : (local variable) given weight for element e_i.

LINK$_{ij}$: (local variable) true if link from e_i to e_j exists in BG(I).

The followings are global variables.

SOURCE$_i$: true if e_i is a source.

SINK$_i$: true if e_i is a sink.

T_i : tree with e_i as its root.

AUGT : the tree that contains the augmenting path.

AUG : the sink in the augmenting path.

Δ_{fs} : the incremental weight of the augmenting path.

If e_j is in T_i then e_j will have the following variable values:

(1) PARENT(i,j) = parent index of e_j in T_i

(2) LEVEL(i,j) = level number of e_j in T_i

(3) $\Delta(i,j)$ = temporary incremental weight of path from e_i to e_j

If e_j is not in T_i then P(i,j)=null and $\Delta(i,j)$=$-\infty$.

Tree T_i also has two variables:

(1) SINK(T_i) = the sink with greatest $\Delta(i,j)$ among all sinks in T_i.

(2) SINKLEVEL(T_i) = the level of SINK(T_i).

(3) $\Delta_s(T_i)$ = incremental weight from i to SINK(T_i).

If T_i does not contain a sink then SINK(T_i) is null, SINKLEVEL(T_i) is ∞, and $\Delta_s(T_i)$ is $-\infty$.

In the initialization for Step 3, we build a tree T_i for each element e_i with root = i. So, element e_i is at level 0. $\Delta(i,i)$ is evaluated and if e_i is a sink then $\Delta_s(T_i)$ is $\Delta(i,i)$ and SINK(T_i) is i. Then each son, e_j, of e_i in BG(I) is added to level one of T_i with *PARENT*(i,j) set to be i at line 5. $\Delta(i,j)$ values are evaluated at line 6. If one of the sons, say e_j, is a sink then SINK(T_i) will be j, $\Delta_s(T_i)$ is $\Delta(i,j)$, and SINKLEVEL(T_i) is 1. If more than one sink exists for T_i, the write-max at line 7 will pick the sink with the greatest $\Delta(i,j)$ value.

The q-th iteration in the algorithm corresponds to the q-th stage of the tree search For each tree T_i, each leaf e_j at level LEVEL$(i,j) = 2^q$ is examined. Each node in T_j will try to add itself to T_i. If several nodes representing the same element e_k are attempting this simultaneously, then the write-max at line 8 will pick the node with the greatest $\Delta(i,k)$ value. If more than one node representing e_k at different levels of T_i has the same greatest $\Delta(i,k)$ value, than the one at the lowest level (closest to the root) is chosen by the write-min at line 9. If some of the newly added nodes of T_i are sinks, then the sink e_j with maximum $\Delta(i,j)$ value is chosen by the write-max in line 11 and $\Delta_s(T_i)$ becomes $\Delta(i,j)$. The sink with this maximum Δ value at the lowest level is chosen by the write-min at line 12. SINKLEVEL(T_i) and SINK(T_i) are assigned accordingly at line 13.

If the root of tree T_i is a source and it contains a sink, then a source-sink path has been found. If more than one such path is found during the same iteration, then the one giving a maximum incremental weight of $\Delta_s(T_i)$ is chosen by the write-max at line 14. The tree with the shortest source-sink path with this $\Delta_s(T_i)$ value is chosen by the write-min at line 15 as AUGT and AUG is the sink in this path.

For the weighted algorithm, we must exhaust all source-sink paths to determine the one with maximum incremental weight. A tree will remain active until no element exists at the 2^{q+1} level.

Algorithm 4.6 : Fast Weighted Algorithm

Steps 0,1, and 2 are the same as Algorithm 4.4.
Step 4 will be shown in section 4.4.3.

Step 3 now consists of the following iterations:
Processors P_{ijk} will execute the following code.

Initialization :

```
1.    for each element $e_i \in E$
2.        build $T_i$ with $i$ as its root:
          PARENT$(i,i) \leftarrow^* 0$;  LEVEL$(i,i) \leftarrow^* 0$
          if $i \in I$ then $\Delta(i,i) \leftarrow^*$ -WEIGHT(i) else $\Delta(i,i) \leftarrow^*$ WEIGHT(i)
3.        if $SINK_i$ then
              $\Delta_s(T_i) \leftarrow^* \Delta(i,i)$;   SINK$(T_i) \leftarrow^* i$;  SINKLEVEL$(T_i) \leftarrow^* 0$
4.        for each element $e_j \neq e_i$
              if $LINK_{ij}$ then
                  PARENT$(i,j) \leftarrow^* i$; LEVEL$(i,j) \leftarrow^* 1$
5.                if $j \in I$ then $\Delta(i,j) \leftarrow^* WEIGHT(i) - WEIGHT(j)$
                                 else $\Delta(i,j) \leftarrow^* WEIGHT(j) - WEIGHT(i)$
6.            if $e_j$ is a sink and $\Delta(i,j) > \Delta_s(T_i)$ then
                  write-max( $\Delta_s(T_i)$,$\Delta(i,j)$ )
              <synchronize>
                  if $\Delta_s(Ti) = \Delta(i,j)$ then
                      SINK$(T_i) \leftarrow^* j$; SINKLEVEL$(T_i) \leftarrow^* 1$
          if no element appears at level 2 then $T_i$ is inactive

q ← -1
repeat the following until DONE
q-th iteration :
q ← q + 1
7.   for each active $T_i$
         for each $e_j$ at level LEVEL$(i,j) = 2^q$
           (* $T_j$ is examined *)
8.         for each $e_k$ in $T_j$
              write-max($\Delta(i,k)$, $\Delta(i,j) + \Delta(j,k)$)
           <synchronize>
9.            if $\Delta(i,k) = \Delta(i,j) + \Delta(j,k)$ then
                 write-min( LEVEL$(i,k)$, LEVEL$(i,j) + 2^q$ )
              <synchronize>
10.              if LEVEL$(i,k)$ = LEVEL$(i,j) + 2^q$ then
                    PARENT$(i,k) \leftarrow^{**}$ PARENT$(j,k)$
11.           if SINK$(T_j) \neq$ null then
                 write-max($\Delta_s(T_i)$, $\Delta_s(T_j) + \Delta(i,j)$ )
              <synchronize>
12.              if $\Delta_s(T_i) = \Delta_s(T_j)$ then
                    write-min( SINKLEVEL$(T_i)$, SINKLEVEL$(T_j) + 2^q$ )
                 <synchronize>
13.                 if SINKLEVEL$(T_i)$ = SINKLEVEL$(T_j) + 2^q$ then
                       SINK$(T_i) \leftarrow^{**}$ SINK$(T_j)$
14.      if i is a source and SINK$(T_i) \neq$ null then
             write-max( $\Delta_{fs}$, $\Delta_s(T_i)$ )
         <synchronize>
15.        if $\Delta_{fs} = \Delta_s(T_i)$ then write-min( SL, SINKLEVEL$(T_i)$ )
           <synchronize>
16.           if SL = SINKLEVEL$(T_i)$ then $AUGT \leftarrow^{**} T_i$
              <synchronize>
17.              if $AUGT = i$ then $AUG \leftarrow$ SINK$(T_i)$
         if no element exists at level $2^{q+1}$ then
```

```
        $T_i$ becomes inactive
    if there is no active tree then DONE is true.
```

The following lemmas are related to Step 3 of both algorithms 4.5 and 4.6.

Lemma 4.4.1 : A node in T_j will not be added to a tree T_i unless all of its ancestors in T_j are added to T_i.

Proof : First let us assume that no two nodes for the same element can exist at the same level in the full tree for each tree T_i. For the cardinality problem, if a node, n_0, for element x has an ancestor node, n_1, for element a which is not added, then another node, n_2, for element a exists at a lower level than n_1. Then either a node n_3 for element x is inserted into T_i as a descendent of n_2 or a node n_4 for x appears at a lower level than n_3. In the first case n_3 is at a lower level than n_0 and in the second case, n_4 is at a lower level than n_0 and hence n_0 will not be added.

For the weighted problem, if a node n_0 for element x has an ancestor node n_1 for element a which is not added to T_i, then another node n_2 for a appears else where with a greater Δ_{ia} value than node n_1. Then either a node n_3 for x is inserted into T_i as a descendent of n_2 or a node n_4 for x appears still elsewhere with a greater Δ_{ix} than n_3. Then n_3 or n_4 has greater Δ_{ix} value than node n_0 in the first or second case respectively.

In the above arguments, if nodes n_1 and n_2 are at the same level in the full tree of T_i, then we can assume that either one of the nodes is inserted into T_i because all we recognize in building the tree are the parent element identities and the level number or Δ value. Hence there is no distinction between whether element x is added as a descendent of n_1 or n_2. With this reasoning, we can say that the lemma holds also for cases with nodes for the same element at the same level □.

Lemma 4.4.2 : At the q-th iteration, no new nodes can be added as descendents to leaves at levels lower (closer to the root) than level 2^q.

Proof : At the q-th iteration, if a leaf appears at a level lower than 2^q, this means that either it represents a sink, or its descendent node, x, has been checked and could not be added to the tree. In the first case, there exists no descendent of the leaf. In the second case, by Lemma 4.4.1, no new nodes can be attached as descendents of this leaf because the ancestor x is missing. □

Lemma 4.4.3 : A tree which has no node at the 2^{q+1}-th level after the q-th iteration will not acquire new nodes during later iterations even if it remains active.

Proof : Since leaves exist only at levels lower than 2^{q+1}, no new nodes can be added as descendents of any leaf at the $q+1$-th iteration by Lemma 4.4.2. □

Lemma 4.4.4 : The algorithms are correct.

Proof : At any stage of the algorithm, if more than one node exists for the same element in the full tree for a tree T at this stage ("full tree" is defined in Section 4.4.1), then, in the cardinality algorithm, only one node at the lowest level (closest to the root) is retained. For the weighted algorithm, only one node with the maximum incremental weight is retained. By Lemma 4.4.2 the algorithms correctly consider only leaves at the 2^q level at the q-th iteration. By Lemma 4.4.3, it is correct to deactivate trees at the q-th stage if they have no leaf at the 2^{q+1} level . So, Step 3 in either algorithm is a breadth-first search of the border graph.□

Lemma 4.4.5 : There will be at most $\log R$ iterations.

Proof : If a source-sink path exists in BG(I), then the maximum height of the trees before a source-sink path is discovered in the cardinality algorithm or all source-sink paths are searched for the weighted algorithm will be $2R$. If no source-sink path exists in

BG(I), then the maximum height of the tree will also be $2R$ before all trees become inactive because a path with no repeated node can have length at most $2R$. Since the height of these trees has increased exponentially from the original height of one, the number of iterations will be O(log R). □

Theorem 4.4.6 : The search for a source-sink path can be done in O(log R) time using m^3 processors.

Proof : The correctness of the algorithms is proved in Lemma 4.4.4. At the 0-th iteration of both algorithms, we can assign a processor to each of the m possible links in each of the m trees. Hence m^2 processors are required for this iteration. At the q-th (q>0) iteration of both algorithms, we can assign processor PE_{ijk} to handle each active tree T_i, each element e_j at level 2^q of T_i, and each e_k in T_j. There are m possible active trees, each tree has less than m possible leaves at level 2^q and each tree has m possible elements, so m^3 processors will be needed.

Each iteration of both algorithms requires constant time. From Lemma 4.4.5, there are O(log R) iterations, so the time complexity is O(log R). □

Theorem 4.4.7 : For the cardinality problem, the search for a source-sink path can be done in O(log R) time using m^3/log$R$ processors.

Proof : For the cardinality algorithm, the required number of processors as stated in Theorem 4.4.6 can be reduced by using Brent's Theorem. In this algorithm, each node can become a leaf at the 2^q level for at most one q value because it cannot be added twice to a tree. Hence the total number of leaves at the 2^q level for all q is no more than m. Since each such leaf requires O(m) operations when a tree is linked to it, and there are m trees in total, the number of elementary operations is O(m^3). Applying Brent's Theorem, we need m^3/log$R$ processors to achieve the same time bound. To apply Brent's Theorem, we have to solve the problem of job assignment for the processors.

We can determine which elements are at the 2^q level in constant time:

```
$k \leftarrow 0$
if $LEVEL(i,j)=2^q$ then fetch-and-add($k$,1)
$LEAF[k] \leftarrow j$ ;    $TREE[k] \leftarrow i$
$NUMBER-OF-LEAVES \leftarrow k$
```

where LEAF and TREE are arrays of size m.

Then jobs $(m-1)k$ to mk are assigned to hooking tree T_j to tree T_i at the q-th stage for $k=1,2,..NUMBER-OF-LEAVES$ where

$i = TREE[k]$ and $j = LEAF[k]$

□

4.4.3. Backtracking by Recursive Doubling

We will backtrack at every node simultaneously to form pieces of paths that may be part of the augmenting sequence. Each element e_t will be the head of such a broken backward path. These paths will grow exponentially to a maximum length of $\min(m,2R)$. Array B will represent the paths:

B(i,j) is the j-th element in the backward path headed by e_i.

B(i,0) is i itself.

Each processor P_{ij} with $i,j=1,2,...m$ does the following:

Step 4 : (* augmentation *)

```
1.   if $SOURCE_i$ then $PARENT(AUGT, i) \leftarrow 0$
2.   $B(i,0) \leftarrow i$; $B(i,1) \leftarrow PARENT(AUGT, i)$
     $q \leftarrow -1$

3.   while not DONE
         $q \leftarrow *q+1$
4.       if $i=AUG$ and $B(i,2^q)=0$ then DONE
         else
             $p_i \leftarrow PARENT(AUGT, B(i, 2^q))$
             if $1 \leqslant j \leqslant 2^q$ then $B(i, 2^q+j) \leftarrow B(p_i, j)$
         <synchronize>
     end-while
5.   if $i=AUG$ and $j < 2^q$ then
         $k \leftarrow B(i, j)$
         if $e_k \in I$ then $I \leftarrow I - e_k$ else $I \leftarrow I + e_k$
```

Lemma 4.4.8 : Step 4 requires $O(\log R)$ time using m^2 processors.

Proof : The algorithm demands that P_{ij} be used where $i, j=1,2,...m$. Hence m^2 processors are required. The algorithm grows the arrays B from an initial length of 1 to a maximum length of $O(R)$ which is the length of an augmenting sequence. The growth is exponential, so $O(\log R)$ iterations of Step 4.1 are needed. Each iteration requires constant time and a time complexity of $O(\log R)$ is thus achieved. □

Now, if we substitute the time complexities stated in Theorem 4.4.7 and Lemma 4.4.8 into the complexities of Steps 2 and 4 respectively in the time analysis for Algorithm 4.2, we get the following result:

Theorem 4.4.9 : The cardinality two-matroid intersection problem can be solved in $O(R(\log R + c(m)))$ time using $O(m^3/\log R)$ processors.

Similarly, if we substitute the time complexities stated in Theorem 4.4.6 and Lemma 4.4.8 into the complexities of Steps 2 and 4 respectively in the time analysis for Algorithm 4.4, we get the following result:

Theorem 4.4.10 : The weighted two-matroid intersection problem can be solved in $O(R(\log R + c(m)))$ time using $O(m^3)$ processors.

Analysis

In the above algorithms, the time complexities are improved but more processors are used. In fact, the time-processor product will be greater than the sequential time. We cannot reduce the number of processors and maintain $O(\log R)$ time because the number of elementary operations in the algorithm is greater than the sequential time.

For the unweighted algorithm (Algorithm 4.5), the number of elementary operations for Step 3 is $O(m^3)$. The total number of elementary operations is $O(m^3R + mR^2c(m))$. This is greater than the sequential time of $O(mR^2c(m))$. By Brent's Theorem, since the depth is $O(R\log R + Rc(m))$, and assuming $c(m) < \log R$, $O(m^3/\log R)$ processors have to be used to maintain the depth.

For Step 3 of the weighted algorithm (Algorithm 4.6), at each iteration, for each element, $O(m)$ trees will be examined and each tree has $O(m)$ nodes: Hence, in this step, the number of elementary operations is $O(m^3\log R)$. The total number of elementary operations for the entire algorithm will be $O(m^3R\log R + mR^2c(m))$ and this is greater than the sequential time of $O(mR^2c(m))$. By Brent's Theorem, since the depth is $O(R\log R + Rc(m))$, and assuming $c(m) < \log R$, $O(m^3)$ processors have to be used to maintain the depth.

4.5. The case with the original F&A CRCW model

In the original fetch-and-add PRAM model, write-max and write-min are not allowed, and we have shown that simulation of these instructions will take $O(\log m)$ time if m is the number of simultaneous write-max or write-min operations to the same location. The time bounds for some of the algorithms presented in this chapter will be different if the original model is used.

Algorithm 4.2 for cardinality matroid intersection is not affected since no write-max or write-min is used. Algorithm 4.4 for weighted matroid intersection uses the write-max instruction when searching for source-sink paths. Hence, the time requirement of this step becomes $O(R \log m)$ instead of $O(R)$ since there may be $O(m)$ simultaneous write-max's to the same location, and the time complexity for Algorithm 4.4 becomes $O(R^2 \log m + Rc(m))$. There will be $O(mR^2 c(m) + mR^3 \log m)$ elementary operations in this algorithm. It can be shown that Brent's Theorem can be applied and hence we can use mR processors to simulate the algorithm within the same time bound.

The fast algorithms are also affected. Algorithm 4.5 for cardinality matroid intersection uses write-min in Step 3. Algorithm 4.6 for weighted matroid intersection uses write-max in Step 3. Both algorithms will now have a time complexity of $O(R(\log R \log m + c(m)))$. The number of elementary operations in these algorithms becomes $O(m^3 R \log R \log m + mR^2 c(m))$. Assuming $c(m) \leqslant (\log R \log m)$, m^3 processors have to be used to maintain the depth.

5. BIPARTITE MATCHING PROBLEMS

In this chapter, we will derive parallel algorithms for the special case of bipartite matching problems. These problems are defined in example 1.9. Since we have derived parallel algorithms for the general two-matroid intersection problems, we would like to see how well they perform on the matching problems. This is done in the next section. It will be shown that the performance is not very good compared to the sequential times of the best known matching algorithms. This suggests that we could achieve better results by taking advantage of the special features of matching problems when designing parallel algorithms.

In the second section we design parallel algorithms based on one of the fastest known sequential bipartite matching algorithms. The cardinality bipartite matching problem will be transformed into a max-flow problem on a unit network. We have chosen Dinic's Algorithm to solve the max-flow problem. The transformation is shown and Dinic's Algorithm will be described. A parallel algorithm which achieves perfect speed-up over Dinic's Algorithm has been designed in [SV1-82]. We will describe the parallel max-flow algorithm in [SV1-82] and show how to modify it to give simpler and better results. The simplified algorithm achieves perfect speed-up.

The weighted bipartite matching problem is solved by transforming it into the min-cost flow problem. The sequential algorithm for the min-cost flow problem is based on the augmenting path method in which each augmentation consists of solving a shortest-paths problem. The shortest-paths problem is solved using an efficient sequential algorithm due to Dijkstra. A parallel algorithm based on these sequential algorithms is then developed. Applying Brent's Theorem for this parallel algorithm gives a perfect speed-up.

Finally, we can solve the shortest-paths problem using a technique similar to that used to search for a source-sink path in the border graph of the two-matroid problems. We will show how to build the shortest-path tree in logarithmic time in parallel. This results in a fast parallel weighted bipartite matching algorithm. This fast algorithm can be used to solve the cardinality matching problem in sub-linear parallel time.

5.1. Applying the General Algorithms

This section will show how we can use the general two-matroid algorithms in the previous chapter to solve both the unweighted (cardinality) and weighted bipartite matching problems. Step 2, which builds the current border graph, in parallel algorithms 4.2 and 4.4 requires some modification.

Let us state the definition of bipartite matching problem again (see Example 1.9). Let $G=[X,Y,E]$ be a given bipartite graph, where X and Y are two disjoint sets of vertices, and E is a set of arcs $\{i,j\}$ in which $i \in X$ and $j \in Y$. The bipartite matching problems will be special cases of the 2-matroid intersection problems in which the two matroids $M_1=(E,\mathbf{I}_1)$ and $M_2=(E,\mathbf{I}_2)$ are partition matroids. A set $I \subseteq E$ is in $\mathbf{I}_1$ iff no two arcs in I are incident to the same vertex in X. Similarly $I \in \mathbf{I}_2$ iff no two arcs in I are incident to the same vertex in Y. The arcs of the bipartite graph G are the elements of the matroids, and a matching of the graph is an intersection of the matroids.

If two arcs have a common vertex in X then they form a circuit in the matroid M_1. If two arcs have a common vertex in Y, then they form a circuit in matroid M_2. In the following, we assume that we have information about the adjacencies between all pairs of arcs and their common vertices in G. The variable $ADJX_j$ will be the number of arcs in the current intersection I adjacent to arc j with the common vertex in X. $ADJY_j$ will be the number of arcs in I adjacent to arc j with the common vertex in Y. If an arc j in $E-I$ has adjacent arcs in I at both endpoints of it, then $ADJX_j$ and $ADJY_j$ will both be positive, and the arc can be neither a source nor a sink in the border graph BG(I). If it has adjacent arcs in I at one or none of the endpoints, then it will be both a source and a sink.

Step 2 : (* building a border graph *)
Processors P_{ij} , $1 \leq i , j \leq m$, perform the following in parallel.

```
1.      $ADJX_j \leftarrow^* 0$ ; $ADJY_j \leftarrow^* 0$
        if arc$_i \in I$ and arc$_j$ is adjacent to arc$_i$ with common vertex $v \in X$ then
            $LINK_{ij} \leftarrow$ true
2.          fetch-and-add( $ADJX_j$ , 1)
3.      if arc$_i \in I$ and arc$_j$ is adjacent to arc$_i$ with common vertex $v \in Y$ then
4.          $LINK_{ji} \leftarrow$ true
5.          fetch-and-add( $ADJY_j$ , 1)
            <synchronize>
6.      if $ADJX_j \geq 1$ and $ADJY_j \geq 1$ then
            $SOURCE_j \leftarrow^*$ false; $SINK_j \leftarrow^*$ false
        else
            $SOURCE_j \leftarrow^*$ true; $SINK_j \leftarrow^*$ true
```

The above step can be done in constant time with m^2 processors.

For the cardinality bipartite matching problem, more modification is needed to achieve a better time complexity. In [HK-73], it is proved that for a sub-optimal intersection I, there may be more than one shortest vertex-disjoint augmenting path, and if we augment I by these paths simultaneously, then the total number of augmentations required is $O(\sqrt{n})$. Hence, to solve this problem more efficiently, steps 3 and 4 of Algorithm 4.2 must also be modified so that all source-sink paths of the shortest length are found instead of just one. A boolean array $AUG[1..m]$ is used so that if element e_i is a sink in one of the shortest source-sink paths, then $AUG[i]$ is marked true, otherwise it is false. To make paths distinct, we restrict each parent node to have only one son. This is done by marking a new variable SON_i to be j if e_i becomes the parent of element e_j.

ALGORITHM 5.1 : (* unweighted matching *)
Same as Algorithm 4.2 except for the following modifications:
Step 2 is modified as mentioned above
Lines 4 to 8 of Step 3 are modified as follows:

```
4.      while path not found and not $ENDSEARCH$
            $ENDSEARCH \leftarrow^*$ true
5.          if $LINK_{ij}$ and $ACTIVE_i$ and $PARENT_j$ =null then
            <synchronize>
                $ACTIVE_i \leftarrow^*$ false
                $SON_i \leftarrow^{**} j$
            <synchronize>
                if $SON_i = j$ then
6.                  $PARENT_j \leftarrow i$
7.                  if $SINK_j$ then $AUG[j] \leftarrow$ true (path found)
                    else
8.                      $ACTIVE_j \leftarrow$ true; $ENDSEARCH \leftarrow^*$ false
            <synchronize>
        end-while
```

Also, line 3 in Step 4 is modified as:

```
3.          if $AUG[j]$ and $i = PARENT_j$ then
```

If the faster Algorithm 4.5 is used instead of Algorithm 4.2, we do not have to worry about distinguishing the paths because they will be in different trees. We simply use a boolean array $AUGT[1..m]$ in which $AUGT[i]$ is true iff tree T_i contains a shortest source-sink path.

ALGORITHM 5.2 : (* fast unweighted matching *)

Same as Algorithm 4.5 except for the following:

Step 2 is modified as mentioned above.

Line 15 of Step 3 is modified as follows:

```
15.     if SL = SINKLEVEL($T_i$) then $AUGT[i] \leftarrow^*$ true
```

The backtracking by recursive doubling can be done with similar modification.

The weighted matching algorithms are as follows:

ALGORITHM 5.3 : (* weighted matching *)

Same as Algorithm 4.4 except that Step 2 is modified as mentioned above.

ALGORITHM 5.4 : (* faster weighted matching *)

Same as Algorithm 4.6 except that Step 2 is modified as mentioned above.

Time Analysis

The next four results follow from the time analysis of the general algorithms. Note that $c(m)$ is constant because we take constant time to build the border graphs. Also, $O(R)$ is taken to be $O(n)$ because we can have at most $n/2$ arcs in the final matching.

Theorem 5.1.1 : Algorithm 5.1 finds a maximum cardinality matching for a bipartite graph with m arcs and n vertices in $O(\sqrt{n}\,n)$ time using m processors.

Theorem 5.1.2 : Algorithm 5.2 finds a maximum cardinality matching for a bipartite graph with m arcs and n vertices in $O(\sqrt{n} \log n)$ time using $m^3/\log n$ processors.

Theorem 5.1.3 : Algorithm 5.3 finds a maximum weighted matching for a bipartite graph of m arcs and n vertices in $O(n^2)$ time using mR processors.

Theorem 5.1.4 : Algorithm 5.4 finds a maximum weighted matching for a bipartite graph with m arcs and n vertices in $O(n \log n)$ time using m^3 processors.

From the above analysis, we see that the time-processor products of all these algorithms are greater than the sequential time of the best known matching algorithms. The reason is that the special features of the matching problem are not exploited. To achieve better results we will use the fastest known sequential algorithms for these problems.

5.2. ALGORITHMS WITH PERFECT SPEED-UP

5.2.1. Unweighted Bipartite Matching

The unweighted bipartite matching problem can be transformed into the integral maximum flow problem. In fact, the fastest sequential algorithm for unweighted bipartite matching known is based on this transformation [Ta-83]. A parallel algorithm has been designed in [SV1-82] using a different parallel model for the general integral maximum flow problem. First we will define the flow problem, list some properties of network flows, and describe the sequential algorithm in [Ta-83]. Then we show how to make use of the simplicity of the transformed problem and our parallel model to derive a parallel algorithm which is both simpler and faster than the algorithm in [SV1-82].

Basics of Network Flow

The following definitions use the terminology of [SV1-82].

Definition 5.2.1 : A *directed flow network* $N = (G, s, t, c)$ is a quadruple, where

(1) $G = (V, E)$ is a directed graph;

(2) s and t are distinct vertices called the source and the sink respectively;

(3) $c : E \rightarrow R^+$ assigns a non-negative capacity $cap(e)$ to each $e \in E$.

A directed flow network is a *0-1 network (unit network)* if $cap(e) = 1$ for all $e \in E$.

Definition 5.2.2 : Let $u \rightarrow v$ denote a directed arc from u to v.

A function $f : E \rightarrow R^+$ is a *flow* if it satisfies:

(1) The *capacity rule*:

$f(e) \leqslant cap(e)$ for all $e \in E$

(2) The *conservation rule*:

$$\mathrm{IN}(f, v) = \mathrm{OUT}(f, v) \quad \text{for all } v \in V - \{s, t\}$$

Where

$$\mathrm{IN}(f, v) = \sum_{u \to v \in E} f(u \to v) \text{ is the total flow entering v.}$$

$$\mathrm{OUT}(f, v) = \sum_{v \to u \in E} f(v \to u) \text{ is the total flow emanating from v.}$$

The flow *value* $|f|$ is $\mathrm{OUT}(f, s) - \mathrm{IN}(f, s)$.

Definition 5.2.3 : A flow f is a *maximum flow* if $|f| \geqslant |f'|$ for any other flow f'. A flow f *saturates* an arc e if $f(e) = cap(e)$. A flow f is a *maximal flow* (a *blocking flow*) if every directed path from s to t contains at least one saturated arc.

Definition 5.2.4 : The *residual graph* R for a flow f is the graph with vertex set V, source s, sink t, and an arc $[v, w]$ of capacity $res(v, w) = cap(v, w) - f(v, w)$ for every arc $[v, w] \in E$ such that $cap(v, w) > f(v, w)$, and an arc $[v, w]$ of capacity $res(v, w) = f(w, v)$ for every arc $[w, v] \in E$ such that $f(w, v) > 0$. An *augmenting path* for f is a path p from s to t in R.

Definition 5.2.5 : Let R be the residual graph for a flow f. The *level* of a vertex v is the length of the shortest path from s to v in R. The *level graph* L for f is the subgraph of R containing only the vertices reachable from s and only the arcs $[v, w]$ such that $level(w) = level(v) + 1$.

5.2.1.1. Transformation into Max-Flow Problem

The Unweighted bipartite matching problem can be transformed into the integral maximum flow problem in the following way.

Let $G=[X,Y,E]$ be the given undirected graph with vertex set $V=X \bigcup Y$ such that each arc in E has one end in X and the other in Y. We shall denote a typical arc by $\{x,y\}$ where $x \in X, y \in Y$. Let s and t be two new vertices. Construct a graph G' with vertex set $V \bigcup \{s,t\}$, source s, sink t, and capacity one arcs $[s,x]$,$[y,t]$,and $[x,y]$ for every $\{x,y\} \in E$. G' is a unit network. A matching for G of size $|F|$ can be derived from an integral flow f for G' by taking the set of arcs $\{x,y\}$ such that $[x,y]$ has flow one. Hence we can find a maximum cardinality matching for G by solving an integral maximum flow problem on G'.

Dinic's Algorithm

An integral maximum flow can be found using *Dinic's Algorithm.*

Dinic's Algorithm starts with a zero flow and repeat the following step until t is not in the level graph for the current flow.

BLOCKING STEP (Dinic). Find a blocking flow f' in the level graph for the current flow f. Replace f by the flow $f+f'$ defined by $(f+f')(v,w)=f(v,w)+f'(v,w)$.

Theorem 5.2.6 : On a unit network, Dinic's Algorithm halts after at most $2\lfloor\sqrt{n-2}\rfloor$ blocking steps.

Proof : [Ta-83], Pg 102, Theorem 8.5.

Theorem 5.2.7 : On a unit network, Dinic's Algorithm finds a blocking flow in O(m) time and a maximum flow in $O(\sqrt{n}\,m)$ time.

Proof : [Ta-83], Pg 104, Theorem 8.8.

5.2.1.2. Parallel Max-Flow Algorithms

In [SV1-82] an $O(n^2 \log n)$ parallel maximum flow algorithm is designed for the general network problem using m/n processors and a CRCW PRAM model that allows concurrent writes only if the processors attempt to write the same value.

The algorithm in [SV1-82] is a parallel version of Karzanov's Algorithm [Ka-74] which improves on Dinic's Algorithm. The basic idea of the algorithm is quite simple. The algorithm consists of blocking steps. Each blocking step is divided into *pulses*. In the first pulse the source s from it. At the beginning of each of the successive pulses there will be a set of *balanced* vertices (for which $\mathrm{IN}(f, v) = \mathrm{OUT}(f, v)$), and a set of *unbalanced* vertices satisfying $\mathrm{IN}(f, v) > \mathrm{OUT}(f, v)$. The balanced vertices remain idle during the pulse while the unbalanced vertices try to push forward as much of the excess flow as possible. If they cannot eliminate all the excess flow this way, they return the rest backward. Returning the flow backward is done in a "last in first out" order.

It is stated in [SV1-82] that a maximum matching in a bipartite graph can be found by their algorithm within a depth of $O(n^{1.5} \log n)$ time using m/n processors. The algorithm can, in fact, be simplified for the unit network using our parallel model.

Parallel Max-Flow Algorithm for Unit Networks

Due to the simplicity of the unit network, we can eliminate many of the complex structures used in the algorithm in [SV1-82]. In the following algorithm, Step 0 is for initialization. Step 1 and Step 2 together form a pulse. Step 1 will try to push the flows forward, while Step 2 will return backwards the flows which cannot be pushed through in Step 1.

The number of sons, $NUMBER\_OF\_SONS(i)$, of each node, i, in the network is assumed to be known. These sons are assumed to be in an array $SON(i, j)$ for $1 \leq j \leq NUMBER\_OF\_SONS(i)$. In Step 1, each unbalanced node has an incoming flow (unit flow) and no outgoing flow. Each unbalanced node will try to push the flow to one of its sons.

If this son already has a flow or more than one parent is trying to push a flow to it, then it will return the flows back to all but one of these parents in Step 2. Each parent will try each son in turn using the index ordering until either it successfully pushes its flow forward, or it is unsuccessful with all its sons and has to return the flow backwards.

Data Structures used:

$NUMBER\_OF\_SONS(i)$	-- the number of sons of vertex i
$SON(i,j)$	-- the jth son of e_i in the level graph
$INDEX$	-- pointer to $SON(i,j)$ for local vertex i (local variable)
$BALANCED_i$	-- true if and only if vertex i is balanced
$PARENT_i$	-- the vertex from which there is a flow to e_i
$TERM(i)$	-- true if there is a flow from vertex i to sink t

Algorithm 5.5 : Parallel Algorithm for Finding a Blocking Flow on a Unit Network

Input : a level graph for the current flow.
each node i has sons $SON(i,j)$ where $1 \leqslant j \leqslant NUMBER\_OF\_SONS(i)$.
Output : a blocking flow in the level graph.

Processors P_i for i=1,2,...n will perform the following steps.

```
Step 0 : (* Initialization *)
1.          $INDEX \leftarrow 0$ ; $PARENT_i \leftarrow$ null
2.          $BALANCED_i \leftarrow$ true ; $TERM(i) \leftarrow$ false
3.          if i < $NUMBER\_OF\_SONS(s)$ then
                 $PARENT_{SON(s,i)} \leftarrow s$ ; $BALANCED_{SON(s,i)} \leftarrow$ false

Step 1 : (* push *)
4.          if not $BALANCED_i$ then
                 $INDEX \leftarrow INDEX + 1$
5.               if $SON(i, INDEX) = t$ then
                      $TERM(i) \leftarrow$ true ; $BALANCED_i \leftarrow$ true
                 else
6.                    if $PARENT_{SON(i,INDEX)}$ = null then $PARENT_{SON(i,INDEX)} \leftarrow^* i$
      <synchronize>
```

```
Step 2 : (* return *)
7:          if not $BALANCED_i$ then
8.              if $PARENT_{SON(i,INDEX)} = i$ then
                    $BALANCED_i \leftarrow$ true
                    (* node i has successfully pushed its flow forward *)
                else
                    if $INDEX \geqslant NUMBER\_OF\_SONS(i)$ then
                        (* it has tried all of its sons *)
9.                      $j \leftarrow PARENT_i$
10.                     if $j \neq s$ then $BALANCED_j \leftarrow$ false
11.                     $PARENT_i \leftarrow$ null ; $BALANCED_i \leftarrow$ true
                        (* node i has returned the flow back to its parent *)
12.     Go to Step 1 if there are any active nodes (i.e., unbalanced vertices).
```

When the algorithm terminates, the following arcs will have a flow of one:
$(PARENT_i, i)$ for all i, where $PARENT_i \neq$ null;
(i, t) if $TERM(i)$ is true.

Proof of Correctness

We have to show that Algorithm 5.5 does what the algorithm in [SV2-82] does for unit networks.

In the "push" stage of a pulse, the unbalanced vertices should push forward as much excess flow as possible. This is done in Step 1 of Algorithm 5.5. Here, each unbalanced vertex i will try to push all its excess flow (which always has value one) forward to its next son, $SON(i, INDEX)$, at line 5. If this son is the sink t, then we mark $TERM(i)$ to be true and i becomes balanced.

In the "return" stage, if an unbalanced vertex cannot eliminate its excess flow, then it should return the rest backward. This is done in Step 2. Here a vertex is not balanced if and only if it has pushed some flow to one of its sons, j, and the son is not the sink. There may be more than one parent pushing its flow towards this son. Only one will succeed in becoming $PARENT_j$, the parent of j in the blocking flow. At line 8, we identify the successful parent i, and mark i as balanced. If an unsuccessful parent i has no more sons that it can try pushing its flow, then the flow is returned to $PARENT_i$. Then i becomes balanced and $PARENT_i$ becomes unbalanced. □

Time Analysis

Algorithm 5.5 is a special case of the algorithm in [SV2-82] for which the following theorem holds:

Theorem 5.2.8 : The algorithm terminates after at most $2n$ pulses.

Proof : [SV2-82]. Theorem 7.1.

Theorem 5.2.9 :

(1) The unweighted bipartite matching problem can be solved in $O(n\sqrt{n})$ time using m/n processors.

(2) The unweighted bipartite matching problem can be solved in $O(\frac{n^2\sqrt{n}}{p})$ time using p processors where $p \leqslant \frac{m}{n}$.

Proof : Algorithm 5.5 is a simplified version of the algorithm in [SV2-82] and the same arguments for reducing the number of processors hold. Hence, the algorithm requires m/n processors. By Theorem 5.2.6 there will be $O(\sqrt{n})$ blocking steps and by Theorem 5.2.8 there will be at most $2n$ pulses. Since each pulse of Algorithm 5.5 takes constant time, we conclude that the cardinality problem can be solved in $O(n\sqrt{n})$ time using m/n processors.

The total number of elementary operations in each pulse is $O(n)$ since at most n nodes can be unbalanced. By Theorem 5.2.8, there will be at most $2n$ pulses in each blocking step, and by Theorem 5.2.6, there will be $O(\sqrt{n})$ blocking steps. Therefore, the total number of elementary operations is $O(n^2\sqrt{n})$. Applying Brent's Theorem gives part (2) of Theorem 5.2.9. □

The time-processor product for this parallel unweighted bipartite matching algorithm is $O(\sqrt{n}\,m)$. Since this is the same as the sequential time for Dinic's Algorithm on a unit network, a perfect speed-up is achieved.

5.2.2. WEIGHTED BIPARTITE MATCHING

In this section, we present a parallel algorithm for the weighted bipartite matching problem which achieves perfect speed-up. This algorithm is based on a transformation of the matching problem into the minimum-cost flow problem. The sequential algorithm for the minimum-cost flow problem uses an augmenting path method in which each augmentation consists of solving a shortest-paths problem. We will first describe the problem transformation and the sequential algorithm from [Ta-83], and then present the parallel algorithms.

Minimum-Cost Flow

Definition 5.2.10 : Let G be a network such that each arc (v,w) has a cost per unit flow, $cost(v,w)$, in addition to a capacity. Assume that for each arc (w,v) in E, $cost(v,w) = -cost(w,v)$. The *cost* of a flow f is $cost(f) = \Sigma_{f(v,w)>0} cost(v,w) f(v,w)$. A flow is *minimum cost* if among all flows of the same value it has minimum cost. The *minimum-cost flow problem* is that of finding a maximum flow of minimum cost.

Definition 5.2.11 : The *cost* of a path is the sum of its arc costs. The residual graph R for a flow f is defined as in Definition 5.2.4 with the extension that $cost(v,w)$ is the same in R as in G.

5.2.2.1. Transformation into Minimum-Cost Flow

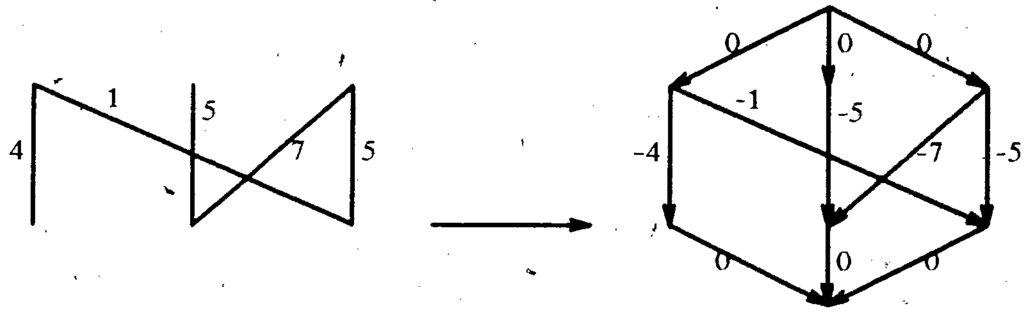
The weighted bipartite matching problem can be transformed into the minimum cost integral flow problem. The transformation is similar to the transformation of the unweighted bipartite matching problem into the integral network flow problem.

Let $G = [X, Y, E]$ be the given undirected graph each of whose arcs has a real-valued weight, denoted by $weight(v,w)$, and with a vertex set $V = X \bigcup Y$ such that every arc in E has one end in X and the other in Y. We shall denote a typical arc by $\{x,y\}$ where $x \in X$ and $y \in Y$.

Let s and t be two new vertices. Construct a graph G' with vertex set $V \bigcup \{s,t\}$, source s, sink t, and capacity-one arcs $[s,x]$ of cost zero for all $x \in X$; $[y,t]$ of cost zero for every $y \in Y$; and $[x,y]$ of $cost = -weight(x,y)$ for every $\{x,y\} \in E$. The graph G' is a unit network. (i.e., all capacities are 1.)

The following diagram is an example of such a transformation.

Example :



given bipartite graph with weights

transformed network

A matching on G of size $|f|$ and weight = $-\text{cost}(f)$ can be derived from an integral flow f on G' by taking the set of arcs $\{x,y\}$ such that $[x,y]$ has flow one. A minimum cost flow will correspond to a maximum weight matching. Hence we can solve the matching problem on G by solving the flow problem on G'.

5.2.2.2. Sequential Algorithm for Min-Cost Flow

Let us look at the theorems that are the basis for an efficient sequential algorithm for solving the minimum cost flow problem.

Theorem 5.2.10 : If f is a minimum-cost flow, then any flow obtained from f by augmenting along an augmenting path (see Definition 5.2.4) of minimum cost is also a minimum-cost flow.

Proof : [Ta-83], pg 109, Theorem 8.12.

Lemma 5.2.11 : If minimum cost augmentation is used, then successive augmenting paths have non-decreasing cost.

Proof : [Ta-83], pg 110, Lemma 8.4.

Algorithm 5.6 : Sequential Min-Cost Flow Algorithm

The following sequential algorithm is from [Ta-83], Chapter 8. From Theorem 5.2.10, if G has no negative cost cycles, then we can find a minimum cost flow of a given capacity by the *augmenting path method*, if we always augment along a minimum cost path. Starting with the zero flow, this method produces a sequence of at most $n/2$ minimum cost flows of increasing value (where $n = |V|$ and $m = |E|$). From Lemma 5.2.11, if we stop the augmenting algorithm just after the last augmentation along a path of negative cost, we will have a minimum cost flow among all flows.

We can compute successive minimum cost augmenting paths by finding a shortest-path tree rooted at s for the residual graph R (see Definition 5.2.4), where the length of an arc is defined to be its initial cost. We use the path in the tree from s to t as our augmenting path.

Due to the simplicity of the initial graph, the first path is simply the one with the minimum cost arc $e \in E$ and the arcs joining e to s and t. To apply *Dijkstra's Algorithm* for the second and successive augmentations, we must have non-negative lengths in the residual graph. Edmonds and Karp [EK-72] have shown how to achieve this by transforming the lengths after each augmentation:

$$\text{length}(v,w) \leftarrow \text{length}(v,w) + \text{dist}(v) - \text{dist}(w)$$

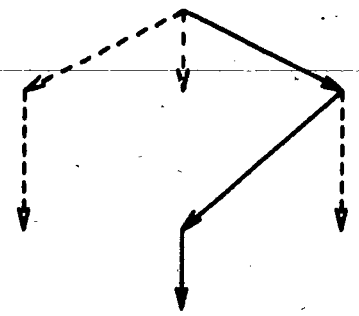
where dist(x) is the length of a shortest path from s to x. The new arc lengths ~~are~~ non-negative.

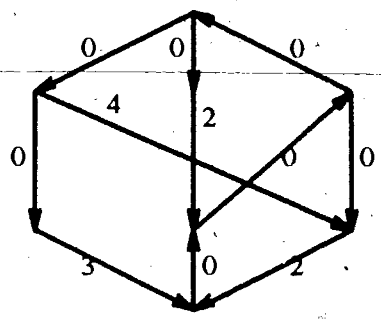
The first augmentation is $O(m)$. Dijkstra's Algorithm is $O(m \log_{(2+m/n)} n)$ and there are at most $n/2$ augmentations. Therefore, the total time complexity is $O(nm \log_{(2+m/n)} n)$.

The following diagram shows the augmenting steps for the example in section 5.2.2.1.

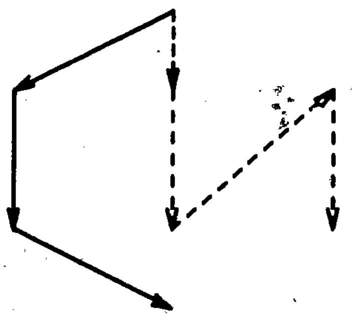
Steps | Shortest-path trees with augmenting paths | Residual graphs with transformed lengths

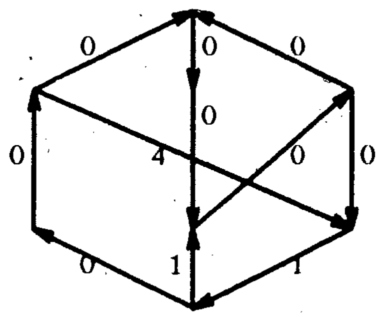
(1)



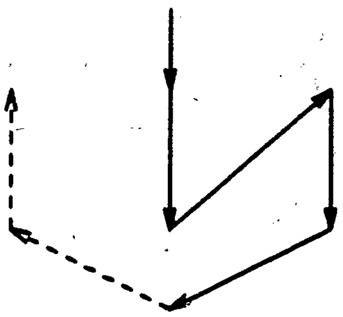


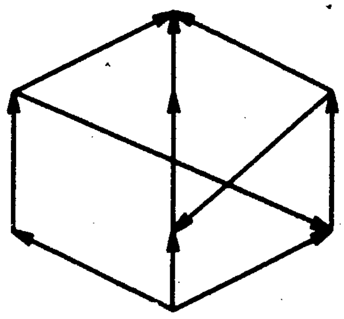
(2)

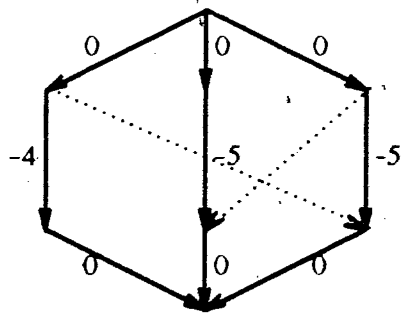




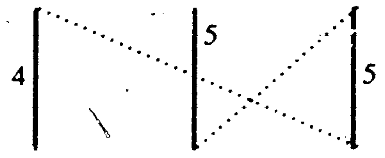
(3)







resultant min-cost flow
(cost=-4-5-5=-14)



corresponding maximum
weight matching
(weight=4+5+5=14)

Dijkstra's Algorithm for Shortest-Path Tree Rooted at s

Starting with the root s, we build a shortest-path tree T arc by arc as follows. We say v *borders* T if $v \notin$ T but some arc is incident to both v and T. A *d-heap* is used to store the vertices bordering T. A d-heap is a complete d-ary tree containing one item per node arranged in heap order: if x and p(x) are a node and its parent, respectively, then the *key* of the item in p(x) is no greater than the *key* of the item in x. The *key* of a vertex v in the heap is the length of the minimum length arc incident to v and T. This arc is stored with v in the heap.

The following step is repeated until all arcs have been considered:

Selection step:

(1) Delete the minimum arc from the d-heap *(deletemin)* and include it in T. This adds a new vertex v to T.

(2) Examine each arc $[v, w]$ incident to v.

(i) if w is not in T and w is not in the d-heap, we *insert* it into the heap.

(ii) if $w \notin$ T but it is in the d-heap with a corresponding arc e of greater value than length$[v, w]$, we replace e by $[v, w]$, modify the key to length$[v, w]$ and *sift-up* to maintain the heap structure.

The running time of each augmentation is thus dominated by the heap operations:

O$(n-1)$ deletemin : O($(n-1)d \log_d n$) time

O$(n-1)$ insert : O($(n-1)\log_d n$) time

O$(m-n+1)$ sift-up : O[$(m-n+1)\log_d n$] time

If $d = \left\lfloor 2 + \frac{m}{n} \right\rfloor$, then the total running time is $O(m \log_{2+m/n} n)$

5.2.2.3. Parallel Algorithm

The parallel algorithm for the weighted bipartite matching problem is similar to the sequential algorithm described above. It is divided into the following steps:

Step 0 : initialization
Step 1 : transform bipartite graph into network
Step 2 : find the first augmenting path
Step 3 : build the residual graph R
Step 4 : find shortest path from source to sink in R
Step 5 : obtain the augmenting path and check its weight

Steps 3 to 5 are repeated until the augmenting path has non-positive weight.

Data Structures for the Parallel Algorithm

WEIGHT (i,j) : is the given weight of arc(i,j) if it exists in the original bipartite graph; and zero otherwise.

LENGTH (i,j) : -*WEIGHT* (i,j) in the first network graph;
$>= 0$ if arc(i,j) exists in the current residual graph;
$-\infty$ otherwise.
(*LENGTH* (i,j) is updated each time a new residual graph is formed.)

Note that *LENGTH* (i,j) as used in the shortest-paths procedure is equivalent to *cost* (i,j) in the minimum-cost network at each augmentation. The shortest path is equivalent to a minimum-cost path.

MIN (i) : is the minimum of *LENGTH* (i,j) for all arcs (i,j) in the residual graph.

M (i) : j if *LENGTH* (i,j) equals *MIN* (i).

PARENT (w) : is the parent node of node w in the current shortest-path tree T.

A (i,j) : 1 if arc(i,j) is in the augmenting path;
0 otherwise.

DIST (w) : is the shortest distance from s to w in the current shortest-path tree T.
(Each time a new node v is included in T, and arc(v,w) exists, we checks if *DIST* (v) + *LENGTH* (v,w) is shorter than *DIST* (w), and update *DIST* (w) if it is.)

MATCHING (i,j) : 1 if arc(i,j) is in maximum matching;
0 otherwise.
(It is updated after each shortest-path tree is found by backtracking from t to s in the tree.)

The following is a description of each step of the algorithm :

(1) In Step 1, the matching graph $G = [X, Y, E]$ is transformed into a flow network. A source node 0 and sink node n+1 are created and directed arcs [0,i], [j,n+1] and [i,j] are included for each arc {i,j} in the original graph where i ∈ X and j ∈ Y. The length (equals cost) of the arc [i,j] will be the negative of its weight. Also, the minimum length of all arcs emerging from each vertex i is found and stored in $MIN(i)$.

(2) In Step 2, the first augmenting path or minimum cost flow is found by looking for the arc with the minimum non-zero length. This will be the minimum of $MIN(i)$ for all vertices i. The augmenting path is a temporary matching and is recorded in the variables *MATCHING* and *A* . The shortest distances from node 0 to each other vertex i are computed and stored in $DIST(i)$.

(3) In Step 3, the residual graph for the current flow is constructed. If a directed arc [i,j] is in the flow (which is equivalent to the augmenting path), then arc [i,j] is deleted by setting $LENGTH(i,j)$ to ∞, and arc [j,i] is added with length zero. For other arcs [i,j], $LENGTH(i,j)$ is modified to $LENGTH(i,j)+DIST(i)-DIST(j)$.

(4) In Step 4, a minimum cost flow of the residual graph is obtained by solving the shortest-paths problem of the graph. This is a simplified version of Dijkstra's Algorithm. Here, a vertex is marked if and only if it borders the temporary tree T being built and a vertex is chosen if and only if it is included in T. The marked vertex i with the shortest *DIST* is chosen at line 5 to be included in T. Line 3 will mark the new vertices which border T after the addition of i. The *DIST* value and parent of such vertices are updated accordingly at line 4.

(5) Step 5 is a simple backtracking from node n+1 through the parents P to the source node 0. The variables *MATCHING* and *A* are updated. Also, the weight of this augmenting path is

calculated in a variable WT. If it is non-positive, then the problem is solved and the algorithm stops.

Algorithm 5.7 : Parallel Weighted Matching Algorithm :

Input : A bipartite graph with n vertices and m arcs. Each arc (i,j) has a given weight $WEIGHT(i,j)$.
Output : A maximum weight matching of G is returned in $MATCHING$.
In the following, processors P_i for i=1,2,...n will perform steps 0 to 4 in parallel. A single processor is sufficient for Step 5.

```
Step 0 : (* initialization *)
1.      $DIST(i) \leftarrow +\infty$ ;    $PARENT(i) \leftarrow$ null
2.      for $j = 0$ to $n$
            $A(i,j) \leftarrow 0$
            $MATCHING(i,j) \leftarrow 0$

Step 1 : (* transformation into network flow *)
1,      MIN $\leftarrow$* 0; $MIN(i) \leftarrow 0$
2.      for $j = 1$ to $n$
        if $WEIGHT(i,j) > 0$ then (* arc(i,j) exists *)
3.          $LENGTH(i,j) \leftarrow -WEIGHT(i,j)$
            $LENGTH(0,i) \leftarrow LENGTH(j,n+1) \leftarrow 0$
4.          if $LENGTH(i,j) < MIN(i)$ then
                $MIN(i) \leftarrow LENGTH(i,j)$ ; $M(i) \leftarrow j$
                (* the minimum arc will be the first augmenting path *)
        else
5.          $LENGTH(i,j) \leftarrow LENGTH(0,i) \leftarrow LENGTH(j,n+1) \leftarrow -\infty$

Step 2 : (* find first augmenting path *)
1.      write-min( $MIN$ , $MIN(i)$ )
        <synchronize>
2.      if $MIN = MIN(i)$ then
            $x \leftarrow$* $i$
        <synchronize>
3.          if $x = i$ then
                $MATCHING(i, M(i)) \leftarrow A(i, M(i)) \leftarrow 1$
                $A(0,i) \leftarrow A(M(i), n+1) \leftarrow 1$
        (* determine DIST(i) *)
        $DIST(i) \leftarrow \infty$
        <synchronize>
4.      for $j = 1$ to $n$ do
5.          if $LENGTH(i,j) \neq -\infty$ then
                if $LENGTH(i,j) < DIST(j)$ then $DIST(j) \leftarrow LENGTH(i,j)$
6.      $DIST(i) \leftarrow 0$ ; $DIST(n+1) \leftarrow$ MIN
```

```
Step 3 : (* build residual graph *)
1.      for j = 1 to n
2.              if A(i,j) = 1 then
                        LENGTH(i,j) ← -∞ ; LENGTH(j,i) ← 0
3.              else
                        if LENGTH(i,j) > 0 then
                                LENGTH(i,j) ← LENGTH(i,j) + DIST(i) − DIST(j)

Step 4 : (* find shortest-path tree from s to t *)
1.      DIST(i) ← ∞
        j ←* 0 ; DIST(0) ←* 0
2.      while n+1 is not chosen do
                j is chosen
3.              if i is not chosen and LENGTH(j,i) ≥ 0 then
                        mark i
4.                      if DIST(j) + LENGTH(j,i) < DIST(i) then
                                DIST(i) ← DIST(j) + LENGTH(j,i)
                                PARENT(i) ← j
                                x ←* ∞
                <synchronize>
5.              if i is marked then write-min( x , DIST(i) )
                <synchronize>
6.              if x = DIST(i) then
                        unmark i ; j ←** i
7.              <synchronize>
        end-while

Step 5 : (* augmentation *)
1.      k ←* 1 ; WT ←* 0
        (* retrieve path by backtracking *)
2.      j ←* n+1
3.      while j > 0 do
                A(PARENT(j),j) ←* 1;
4.              if MATCHING( PARENT(j),j ) then
                        MATCHING( PARENT(j),j ) ←* false
                        WT ←* WT − WEIGHT( PARENT(j),j )
5.              else
                        MATCHING( PARENT(j),j ) ←* true
                        WT ←* WT + WEIGHT( PARENT(j),j )
6.              j ←* PARENT(j)
                <synchronize>
        end-while
7.      if WT ≤ 0 then DONE ←* true
Repeat steps 3 to 5 until DONE.
```

Proof of Correctness

It is not difficult to see that Steps 0,1,2,3 and 5 of Algorithm 5.7 are equivalent to the corresponding steps in the sequential Algorithm 5.6.

Step 4 builds the shortest-path tree. First, line 1 of Step 4 assigns variable j to be the source s. In the following iterations, all sons of j are marked, so the set of marked vertices, V, is the set of vertices bordering T where T is the tree we are building. The shortest distance from source s to each son is calculated at line 4. The marked vertex with minimum distance from s is identified by lines 5 and 6. This vertex is then chosen as the next value of j. Hence, Step 4 is equivalent to Dijkstra's Algorithm and it builds the shortest-path tree. □

Analysis

Theorem 5.2.12 : The weighted bipartite matching problem can be solved in $O(\frac{mn}{p})$ time using p $(\leqslant \frac{m}{n})$ processors on the F&A PRAM parallel model with the addition of the "write-min" operation.

Proof : The time required for each step of the algorithm is $O(n)$. Since the maximum size of the matching is $O(n)$ and the size of the temporary matchings increases by one after each augmentation, the number of augmentations required is $O(n)$. That is , there are $O(n)$ iterations of steps 3 to 5. Hence, the whole process takes $O(n^2)$ time. The number of elementary operations for steps 1 to 4 is $O(m)$, and for Step 5 is $O(n)$. So, the total number of elementary operations is $O(nm)$. The problem of assigning processors to the jobs can be solved easily since at steps 0 to 4 of the algorithm, there are x jobs - one for each of the vertices adjacent to a certain vertex j. At Step 5 there will be only one job at each instant. Hence, the assignment of p $\leqslant$ n processors to these jobs is straightforward. By Brent's Theorem, we can use p processors to implement the algorithm in time $O(\frac{mn}{p})$ if $p \leqslant \frac{m}{n}$. Therefore, we can use m/n processors to execute the algorithm with the same time complexity. □

Theorem 5.2.13 : without the "write-min" operation, Algorithm 5.7 solves the problem in $O(n^2 \log n)$ time using m/n processors.

Proof : We have shown that simulation of the write-min operation requires $O(\log n)$ time on the general PRAM machine. So, the time for Step 4 is $O(n \log n)$ and overall time is $O(n^2 \log n)$. The number of elementary operations for Step 4 also increases to $O(m \log n)$ giving a total of $O(nm \log n)$. Hence, m/n processors are again needed to maintain the time complexity. □

5.3. Fast Algorithms for the Bipartite Matching Problems

As with the general two-matroid intersection problems, the bipartite matching problems can be solved by faster algorithms which use more processors. For the cardinality bipartite matching problem, a sub-linear time bound is achieved. In this section we describe these fast algorithms. The keys to these algorithms are the technique of recursive doubling and a logarithmic time shortest-paths algorithm.

Algorithm 5.8 : Fast Weighted Bipartite Matching Algorithm

In Algorithm 5.7 for the weighted bipartite matching problem, steps 0,1,2, and 3 can be done in $O(1)$ time using n^2 processors. This is because the "for $j = 1$ to n" or "$j = 0$ to n" loops at lines 2, 2, 4, and 1 in steps 0, 1 ,2, and 3 respectively can be done concurrently by assigning n processors, $P_{i1}, P_{i2}, \cdots P \in$ to node i (i.e., one processor for each time through the loop). Backtracking in Step 5 can be done in $O(\log n)$ time by recursive doubling (see Section 4.4.3). We show how to do Step 4 in logarithmic time using the technique developed for Algorithms 4.5 and 4.6 to search for source-sink paths in the border graph.

Step 4 : (* find shortest-path tree *)

Each vertex i in the residual graph has a tree T_i which will grow exponentially to a maximum height of O(n). Each vertex j may appear at most at one node in T_i. If vertex j does appear, then $PARENT(i,j)$ will be the vertex number of its parent node. $LEVEL(i,j)$ will be its level number, and $DIST(i,j)$ will be the shortest distance known so far from vertex i to j in the residual graph. If j is not in T_i then $PARENT(i,j)$ is null ; $LEVEL(i,j)$ is $-\infty$ and $DIST(i,j)$ is ∞.

Step 4 now consists of the following iterations to be done by processors P_{ijk}, $1 \leq i,j,k \leq n$, in parallel.

Initialization :

```
1.      for each vertex $i$ in the graph
                (* build $T_i$ *)
2.              $PARENT(i,i) \leftarrow^* 0$ ; $LEVEL(i,i) \leftarrow^* 0$ ; $DIST(i,i) \leftarrow^* 0$
3.              for each vertex $j \neq i$
                        if $LENGTH(i,j) \geq 0$ then (*$j$ is son of $i$*)
                                $PARENT(i,j) \leftarrow^* i$ ; $LEVEL(i,j) \leftarrow^* 1$ ;
                                $DIST(i,j) \leftarrow^* LENGTH(i,j)$
```

```
$q \leftarrow -1$
The following is repeated until DONE
q-th iteration :
4.      $q \leftarrow q+1$
5.      for each tree $T_i$
6.              for each node $j$ at level $LEVEL(i,j)=2^q$
                        for each node $k$ in $T_j$
                                write-min( $DIST(i,k)$ , $DIST(i,j)+DIST(j,k)$ )
        <synchronize>
7.                              if $DIST(i,k)=DIST(i,j)+DIST(j,k)$ then
                                        $PARENT(i,k) \leftarrow^{**} PARENT(j,k)$

        <synchronize>
        DONE $\leftarrow^*$ true
        <synchronize>
8.      if PARENT(0,$k$) $\neq$ null then
                DONE $\leftarrow^*$ false
```

Time Analysis :

Theorem 5.3.1 : Algorithm 5.8 solves the weighted bipartite matching problem in O(n log$n$) time using n^3 processors.

Proof : Steps 0,1,2,3 of the algorithm can be done in O(1) time using n^2 processors. Step 4 can be done in O(log n) time using n^3 processors as described above. Step 5 can be done in O(log n) time using n^2 processors by recursive doubling (section 4.4.3). There will be O(n) augmentations which implies O(n) iterations of steps 3 to 5. Hence, the overall complexity of the algorithm is O(n log$n$) time using n^3 processors. □

Algorithm 5.9 : Fast Unweighted Matching Algorithm

The unweighted matching problem can be transformed into the weighted matching problem by giving each arc a unit weight. Then Algorithm 5.8 can be modified to solve the unweighted problem.

Theorem 5.3.2 : The unweighted bipartite matching problem can be solved in O($\sqrt{n}$ log$n$) time using n^3/log$n$ processors.

Proof : We can transform an unweighted bipartite matching problem into a weighted bipartite matching problem by giving each arc a unit weight. Each augmentation in the transformed weighted algorithm will be comparable to a blocking step in Dinic's Algorithm. The solution to the shortest-paths problem will contain a maximal set of augmenting paths. This set of paths gives the maximum incremental weight and is equivalent to the set of augmenting paths resulting from a blocking step. By Theorem 5.2.6, there will be O($\sqrt{n}$) blocking steps for Dinic's Algorithm. Therefore we can also say that there will be O($\sqrt{n}$) augmenting steps for the transformed weighted algorithm. The time required for each augmenting step is O(log$n$). When solving the shortest-paths problem, each element in a tree T_i can be a leaf at level 2^q for only one q . Since each such leaf requires O(n) operations when a tree is linked to it, and there are n trees in total, the number of elementary operations is O(n^3). By Brent's Theorem, n^3/log$n$ processors are suffici~~ent to im~~plement the algorithm within the same time bound. Hence we conclude that the unweighted bipartite matching problem can be solved in $O(\sqrt{n} \log n)$ time using n^3 processors. □

Remarks :

Note that we have essentially derived a parallel algorithm for solving the minimum cost flow problem in general. However, it is shown in [Ta-83] that an algorithm based on minimum cost augmentations will require $O(|f|)$ augmentations, where f is the value of the minimum cost flow. The complexity of the parallel algorithm will therefore be $O(|f| \log n)$ which is not *strongly polynomial*. An algorithm for this problem is strongly polynomial if the time complexity is polynomial in the number of nodes and is independent of both costs and capacities. We expect a strongly polynomial parallel algorithm to be found in the future for the general minimum cost flow problem.

5.3.1. About the Shortest-Path Problems

The shortest-paths problem is concerned with a directed network in which arcs may have positive, zero or negative lengths, as long as there are no negative length cycles. Parallel algorithms for two kinds of shortest-path problems are reported below. The two problems are:

(1) The single-source problem - Finding the shortest path from a specified vertex to all other vertices in a network.

(2) The all-pairs problem - Finding the shortest path between every pair of vertices in a network.

Some reported work on parallel shortest-paths algorithms includes [Ar-75], [De-80], [DPL-80], [Ku-82], [MD-81], [Pr-83], [Qu-83] and [Yo-83]. The following table list some of the results of this work.

Table. Parallel All-Pairs Shortest-Paths Algorithms

Reference	Model	Complexity	Processors
[LK-72]	Systolic array	$O(n)$	n^2
[Ar-75]	MIMD-TC	-	-
[Sa-77]	SIMD-SM-R	$O(\log^2 n)$	$n^3/\log n$
[De-80]	MIMD-TC	$O(n^3/p + pn)$	$p << n$
[De-81]	SIMD-PS, SIMD-CC	$O(\log^2 n)$	n^3
[Ku-82]	SIMD-SM-RW	$O(\log n)$	n^4

In this thesis, parallel procedures that solve these shortest-paths problems can be derived as special cases of the main algorithms.

The single-source problem is a sub-problem of the weighted bipartite matching problem which is solved by Algorithms 5.7 and 5.8. In the cardinality two-matroid intersection algorithms (Algorithms 4.2 and 4.5) given in the previous chapter, the sub-problem of finding a shortest source-sink path in BG(I) is a generalization of an all-pairs shortest-paths problem; the lengths of the directed arcs in BG(I) are taken to be one, and only the shortest paths from the sources to the sinks are significant. Similarly, in the weighted two-matroid intersection algorithms (Algorithms 4.4 and 4.6), the sub-problem of finding a maximum weight source-sink path in BG(I) is also an all-pairs shortest-paths problem. The length of each directed arc in the graph in the all-pairs problem is the negative of the augmenting weight of the arc in the border graph.

From the results in this thesis, we are able to solve the all-pairs shortest-paths problem in $O(\log n)$ time using n^3 processors using our model. This is the same as the fastest known parallel time and the time-processor product is smaller by a factor of $\log n$. This is because of the "write-min"/"write-max" operations which seem to be very useful in weighted problems.

Now we will show that the fast procedures for the shortest-paths problems in Algorithms 4.5 and 4.6 are equivalent to "the repeated plus-min method" used by many parallel shortest-paths algorithms. The following is a description from [QD-84] of the repeated plus-min multiplication method used to solve the all-pairs problem in parallel.

Given an n-vertex weighted graph G, the goal is to produce an nxn matrix A such that a_{ij}

is the length of the shortest path from i to j in G. Let a_{ij}^{k} denote the length of the shortest path from i to j with at most k intermediate vertices. Since there are no negative weight cycles in G, $a_{ij}=a_{ij}^{n}$. $a_{ii}^{0}=0$, for all i, $1 \leq i \leq n$, and for all distinct i and j, a_{ij} is the weight of the arc from i to j; if no such arc exists, then $a_{ij}^{0}=\infty$. It is not hard to show that $a_{ij}^{k}=\min\{a_{im}^{k/2}+a_{mj}^{k/2}\}$. Hence, A^{n} may be computed from A^{0} by repeated plus-min multiplication. The fast matrix multiplication algorithms devised by Dekel et al [DNS-81] for the SIMD-CC and SIMD-PS models (see section 2.1) can solve this problem in O(log$n$) time using n^{3} processors.

In our algorithms (Algorithms 4.5, 4.6 and 5.8), a tree T_i for element e_i at the q-th stage corresponds to row i in matrix A^{k}. That is, a node for element e_j in T_i at the q-th stage corresponds to a_{ij}^{k} in the repeated plus-min method.

6. DIRECTED SPANNING TREES

The Directed Spanning Tree (DST) problem (see Example 1.10) is defined as follows:

Given : an arc-weighted directed graph $G = (V, A)$ br

with a distinguished root node with in-degree zero

Find : a maximum weight spanning tree directed from the root node

This is a two matroid intersection problem with the following two matroids:

(1) graphic matroid of G.

(2) partition matroid in which sets of arcs, no two of which are directed into the same node, are independent.

We can apply the general weighted two-matroid intersection algorithms to solve this problem. However, when we build the border graph for this problem, we have to look for the circuits in each of the two matroids. For the partition matroid, each pair of arcs going into the same node is a circuit and this is easy to find. For the graphic matroid, the cycles in the graph are the circuits and the search for circuits is not as easy. Moreover, we have a sequential directed spanning tree algorithm which runs much more efficiently than the general sequential matroid algorithm. We have derived a parallel algorithm which achieves perfect speed-up with respect to this efficient algorithm. We will not show the use of the general parallel algorithm because it will be inferior to this special parallel algorithm. In the following, we describe the sequential directed spanning tree algorithm followed by the parallel version of it.

6.1 Sequential Algorithm

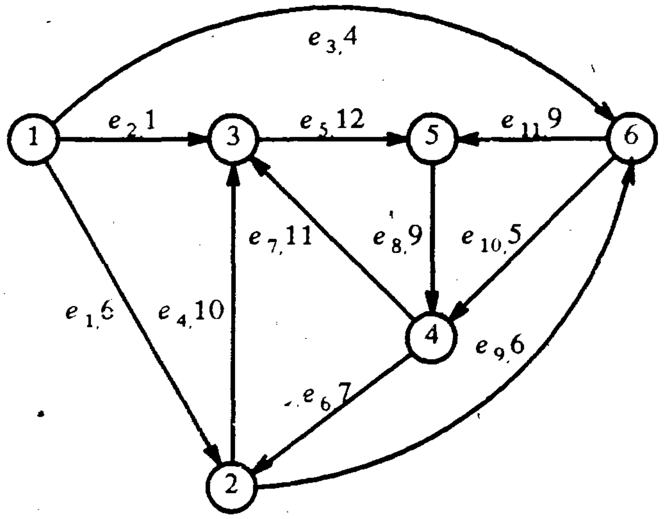
A particularly simple and elegant procedure has been devised by Edmonds [Ed-68] for this special case of weighted matroid intersection. The algorithm has two phases. In the first phase, cycles are replaced one at a time by "pseudo-nodes". In the second phase, pseudo-nodes are expanded in the reverse order.

Algorithm 6.1 : Sequential DST Algorithm

```
    (G is a digraph and w is a weight function on the edges of G.)
    (Phase 1 : contract cycles)
1.  repeat until done
2.      Use greedy algorithm for the partition matroid (i.e. choose maximum weight arc
        entering each node)
3.      if there are no cycles then done
        else
4.          Replace nodes on each cycle by a pseudo-node. Remove all self-loops.
            Replace arcs to or from nodes on the cycle by arcs to or from the pseudo-node.
5.          Replace weights on arcs entering the pseudo-node as follows : if $(i,j)$ is
            replaced by $(i,k)$, where $k$ is a pseudo-node and $i$ is not in the cycle replaced
            by $k$, then set $w_{ik} = w_{ij}$ − (weight of the unique arc of the cycle into
            $j$) + (minimum weight of an arc on the cycle).
    (Phase 2 : Expand pseudo-nodes and choose tree edges)
6.  Choose arcs from the final contraction of the primal algorithm.
7.  repeat until done
8.      Expand a pseudo-node. Choose all arcs except the single arc on the cycle that
        will cause two arcs to enter a single node or pseudo-node.
```

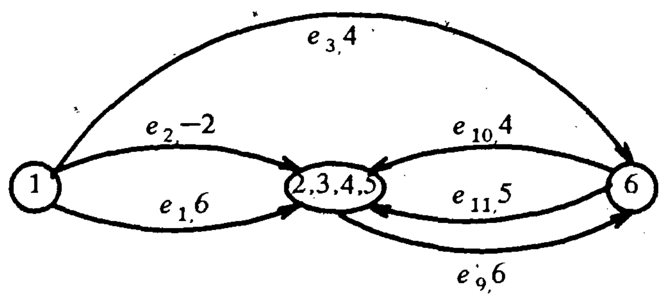
Example

We illustrate this algorithm on an example :

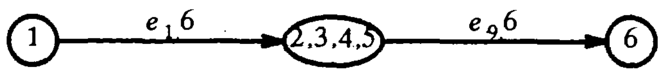


In the first iteration of phase 1, we choose arcs e_5, e_6, e_7, e_8, and e_9 after applying the greedy algorithm on the partition matroid. So, as e_5, e_7, and e_8 form a cycle, we coalesce the corresponding nodes to form a pseudo-node (3,4,5) and change the weights on the arcs as given by the algorithm.

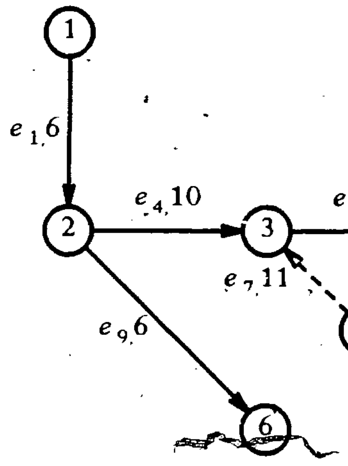
In the second iteration we again apply the greedy algorithm on the partition matroid which forces us to choose e_4, e_6 and e_9. As e_4 and e_6 form a cycle on node 2 and pseudo-node (3,4,5), we coalesce these nodes to form the pseudo-node (2,3,4,5) and change the weights on the arcs. The result is the following graph.



Now, in the third iteration, we choose arcs e_1 and e_9. Since there are no more cycles, phase 1 is finished and e_1 and e_9 are chosen to be in the spanning tree (in line 8). The situation is as follows.



In phase 2, we first expand the pseudo-node (2,3,4,5) and remove the edge e_6 as both edges e_1 and e_6 enter node 2 and e_6 is in the cycle. Arc e_4 is added to the solution. Now, we expand the pseudo-node (3,4,5) and we get:



As e_7 and e_4 both enter node 3, we remove e_7 which is in the cycle and add e_5 and e_8 to the solution. The algorithm ends now and the total weight of the tree so formed is 43.

The correctness proof of the algorithm can be found in the original paper by [Ed-68]. [Ka-71] also proved correctness using a more elegant technique. [Ta-77] presented an efficient implementation for this algorithm which runs in $O(\min\{m \log n, n^2\})$ time.

Algorithm 6.1 finds a maximum weight spanning tree, but the maximum weight intersection may be a forest. Minor modifications to the algorithm to get a maximum weight intersection are described in [La-76], Ch. 8.

6.2 Parallel Algorithm

The parallel algorithm closely follows the sequential algorithm. The following data structures are used in the parallel algorithm :

$A(j,i)$: -1 if there is an edge from j to i in the given graph;
k if there is an edge from j to i but i is a pseudo-node and k is the sub-node in

this pseudo-node to which the edge was formerly directed;
0 otherwise.

WEIGHT (j,i) :
the weight of the edge from j to i if it exists;
$-\infty$ otherwise.

EDGE−ID (j,i) :
the id of the edge from j to i if it exists;
0 otherwise.

$PARENT_i$: j if arc (j,i) is the heaviest (of maximum weight) incoming arc for node i;
null if no such arc exists.

PSNODE : the index of the current pseudo-node .

PSN : used to mark a certain node in a cycle.

PS (i,j) : true if node j is part of pseudo-node i;
false otherwise.

MINWT : the weight of the smallest weight edge in the current cycle.

$X(j)$: 1 if an arc (i,j) has been selected for output where j is inside an pseudo-node and i is not (this implies that the edge coming into j in the cycle will be discarded on expansion);
0 otherwise.

The parallel algorithm consists of 3 phases:

(1) Step 0 initializes the adjacency matrix A, the weights, the edge-id's, and other variables. Step 1 applies the greedy algorithm to the partition matroid, choosing the maximum weight arc (i,j) entering each node j. Node i is marked as the parent of j.

(2) Iterations of Step 2 will continue contracting cycles into pseudo-nodes until all cycles are removed. The search for cycles is done in parallel in lines 2 to 5. Each node will search through its descendents one by one. If one of the descendents points back to the original node, then a cycle has been found. If a cycle is found then the nodes inside this cycle will be contracted into a pseudo-node, x. The PS(x,i) value for each such node i will is marked true. These nodes become "inactive" and the pseudo-node becomes "active". Only active nodes will be considered during the search for cycles. Lines 9 to 11 will consider the arcs going from nodes inside the pseudo-node to nodes outside. If more than one such arc is directed to the same node y outside, then only the one with greatest weight will be retained

as the arc directed from the pseudo-node to node y. The related variables of A, *EDGE−ID*, *WEIGHT* and *PARENT* are updated. Lines 12 to 15 will consider the arcs going into the nodes inside the pseudo-node from nodes outside. The weights of these incoming arcs are modified in line 12 in the same way as in Step 5 of the sequential algorithm. If more than one such arc comes from the same outside node z, then the one with greatest weight will be retained as the arc directed into the pseudo-node from node z. The corresponding variables are updated. Finally, the incoming arc with greatest weight to the pseudo-node is chosen at lines 14 and 15 by using "write-max". *PARENT* of the pseudo-node is thus determined.

(3) After all cycles are removed, Step 3 will expand the pseudo-nodes in the reverse order of their formation. The matrix PS is used to identify the nodes inside the pseudo-nodes. Note that a node is active in line 6 only if it is a node in the pseudo-node which was expanded in the previous expansion and marked active at line 8. The edges coming into these active nodes are output at line 7.

Algorithm 6.2 : The Parallel DST Algorithm

Each processor P_i for $1 \leqslant i \leqslant 2n$ will perform the following steps.

```
Step 0 : (* initialization *)
1.      $WT \leftarrow^* -\infty$ ;    $PSNODE \leftarrow^* n$
2.      for all incoming edges $(j,i)$ with weight w and edge id = q
3.              $A(j,i) \leftarrow -1$ ;  $WEIGHT(j,i) \leftarrow w$ ;  $EDGE-ID(j,i) \leftarrow q$
4.      $X(i) \leftarrow 0$ ; $PSN(i) \leftarrow 0$ ; $PARENT_i$ = null
5.      $PS(n+i,j) \leftarrow$ false  for $j = 1,2,\ldots n$
6.      mark node $i$ active
```

For $j = 1$ to n do Step 1

```
Step 1 : (* find heaviest incoming arcs *)
1.      write-max ( WT ,WEIGHT (i ,j ) )
2.      <synchronize>
3.      if WT = WEIGHT (i ,j ) then PARENT_j ←* i
4.      WT ← -∞
```

Repeat Step 2 until no cycle is found

```
Step 2 : (* contraction of cycle *)
1.      if i = 1 then PSNODE ← PSNODE + 1

        (* detect cycle *)
2.      if i is active then
                j ← PARENT_i ;
3.              while j ≠ null and j ≠ i do
                        j ← PARENT_j ;
4.              if j ≠ null then (* cycle is found *)
5.                      PSN ←* i
                <synchronize>


6.      If no cycle is found go to Step 4
```

```
        (* collect nodes in cycle *)
7.      if $PSN = i$ then
              $PS(PSNODE, i) \leftarrow$ true
              mark node $i$ as inactive ; mark $PSNODE$ active
              $j \leftarrow PARENT_i$
              $MINWT \leftarrow WEIGHT(PARENT_i, i)$
8.            while $j \neq PSN$ do
              <synchronize>
                    $j \leftarrow^* PARENT_i$
                    if $MINWT > WEIGHT(j,i)$ then $MINWT \leftarrow WEIGHT(j,i)$
                    mark node $j$ as inactive
                    $PS(PSNODE, j) \leftarrow$ true
        (* consider edges from pseudo-node *)
9.                  if $A(j,i) \neq 0$ then
10.                       if $WEIGHT(j,i) > WEIGHT(PSNODE,i)$ then
                                $EDGE-ID(PSNODE,i) \leftarrow EDGE-ID(j,i)$
                                $WEIGHT(PSNODE, i) \leftarrow WEIGHT(j,i)$
11.                       $PARENT_i \leftarrow PSNODE$
        (* modify weights of arcs to PSNODE *)
12.                 if $A(i,j) \neq 0$ then
                          $WEIGHT(i,j) \leftarrow WEIGHT(i,j) - WEIGHT(PARENTi, i)$
                                                 $+ MINWT$
13.                       if $WEIGHT(i,j) > WEIGHT(i, PSNODE)$ then
                                $WEIGHT(i, PSNODE) \leftarrow WEIGHT(i,j)$
                                $EDGE-ID(i, PSNODE) \leftarrow EDGE-ID(i,j)$
                                $A(i, PSNODE) \leftarrow j$
              <synchronize>
14.                 write-max( $WT$, $WEIGHT(i, PSNODE)$ )
              <synchronize>
15.                 if $WEIGHT(i, PSNODE) = WT$ then $PARENT_{PSNODE} \leftarrow^* i$
                    $WT \leftarrow \infty$
        end-while
```

Repeat Step 3 until $PSNODE = n$

Step 3 : (* *expansion and output* *)

```
1.      if $i = 1$ then $PSNODE \leftarrow PSNODE - 1$
        $X(i) \leftarrow 0$
        if $PSNODE = n$ then stop
        <synchronize>
```

```
2.      if $i$ = $PSNODE$ then
            if $PARENT_{PSNODE} \neq$ null then
                $j \leftarrow A(PARENT_{PSNODE}, PSNODE)$
                (* $j$ is the node in cycle $PSNODE$ to which the edge
                from $PARENT_{PSNODE}$ is directed *)
3.              mark $X(j) \leftarrow 1$
4.              output edge $EDGE-ID(PARENT_{PSNODE}, PSNODE)$
5.          mark $PSNODE$ inactive
        <synchronize>
6.      if node $i$ is active then
7.          if $X(i) = 0$ then
                output edge $EDGE-ID(PARENT_i, i)$
                mark $i$ inactive
        <synchronize>
8.      if $PS(PSNODE, i)$ then
            mark node $i$ active
```

Proof of Correctness

The correctness of the sequential algorithm has been proved in [CL-65] and [Ed-67]. We will prove that the parallel algorithm does the same things as the sequential algorithm.

(1) Step 1 chooses the heaviest arc directed into each node.

(2) While there exists some cycle in the graph, Step 2 will contract the cycle and replace it with a pseudo-node. Weights of arcs(i, j) directed from outside into the pseudo-node are modified.

There can be more than one arc going into a pseudo-node and with the write-max instruction at line 4, only the maximum weighted arc is chosen.

(3) Step 3 will keep expanding the pseudo-nodes in the opposite order to which they were formed until all pseudo-nodes are expanded. During expansion, there is a unique arc $[j, i]$ of the cycle whose entry would cause two arcs to be directed into the same node i and this arc should be discarded. At line 3, $X(j)$ is marked so that arc $[j, i]$ will not be output at line 7. □

Time Analysis for the DST Algorithm

Step 0 takes $O(n)$ for initialization

Step 1 takes $O(1)$ time and it is repeated $O(n)$ times for a total of $O(n)$.

Step 2 is the contraction of cycles. We do not know in advance how many nodes there will be in a cycle and how many cycles there will be. However, the total time required is proportional to the total number of nodes involved in all the cycles. This is $O(n)$ because we know that a node will be contracted at most in one cycle.

Step 3 is the expansion of pseudo-nodes. Step 3 itself takes constant time but it will be iterated q times where q is the number of pseudo-nodes. We know that $q = O(n)$ and therefore the time for expansion is $O(n)$.

From above, the overall time complexity is $O(n)$.

Since the depth is $O(n)$ and $2n$ processors are used, the time-processor product for Algorithm 6.2 is $O(n^2)$. Recall that the sequential time is $O(\min\{m \log n, n^2\})$. If the given graph is dense so that $m = O(n^2)$, then we have a perfect speed-up. We have now proved the following theorem:

Theorem 6.1: The directed spanning tree problem can be solved in $O(n)$ time using $O(n)$ processors, where n is the number of vertices.

It can be shown that the number of elementary operations for Algorithm 6.2 is $O(n^2)$ and that Brent's Theorem can be applied to get the following result.

Theorem 6.2: The directed spanning tree problem can be solved in $O(\frac{n^2}{p})$ time using p processors where $p \leqslant n$.

If write-max is not used, then steps 1 and 2 will both require $O(n^2 \log n)$ operations and $O(n \log n)$ time. The overall time will be $O(n \log n)$ and $O(n)$ processors must be used to maintain this depth.

7. CONCLUSIONS

In this thesis, parallel algorithms have been designed for two-matroid intersection problems. We have designed parallel algorithms for both the general problems and two special cases : the bipartite matching problems and the directed spanning tree problem. For all of these problems, parallel algorithms which achieve perfect speed-up over the fastest known sequential algorithms have been designed. Another set of fast algorithms achieves almost linear time for the general problems. For the two special cases, we have designed parallel algorithms that perform better than the general algorithms by exploiting special features of the problems. However, the general algorithms are useful in that they provide upper bounds on time and processor complexity.

The parallel algorithms in this thesis were designed by choosing the most efficient sequential algorithms and making them parallel. Unfortunately, some parts of the sequential algorithms for the two-matroid intersection problems are difficult to parallelize. The construction of the border graphs is the easiest part to do in parallel. Backtracking by recursive doubling requires lookahead concepts. At first it may seem that a breadth-first search of a border graph for a source-sink path is sequential in nature. This is because each level of the search depends on the previous levels, and the whole search tree is exponential in size. By examinating properties of the problem, we see that the searches are, in fact, solving shortest-path tree problems. A logarithmic time algorithm for this search is derived in which the heights of the temporary search trees are recursively doubled, while the number of nodes in a search tree never exceeds m , the number of elements in the matroid. This algorithm can be seen as a graph-based version of the repeated plus-min multiplication method as given in [De-81] and [QD-84].

However, the outer layers of the algorithms, which do augmentations, are still sequential. Each augmentation depends on the previous augmentation, and, unlike the breadth first search in which the whole search tree is implicitly embedded in the border graph, no information about future augmentations (i.e., the future border graphs) can be deduced explicitly or implicitly. We do not know if augmentations can be done in parallel, but it looks like a very difficult problem.

There are two possible directions for further research on parallel matroid intersection problems. The first is to keep looking for more efficient parallel algorithms. If the augmentations can be done in parallel, then logarithmic or sublinear time parallel algorithms for the two-matroid intersection problems should result. One possible starting point is to look at sequential algorithms other than those used in this thesis. Although we have chosen the fastest and simplest sequential algorithms, and believe that the other algorithms will not give better results, the possibility has not been ruled out. Also, there may be entirely new strategies that have not yet been discovered for solving the problems in parallel.

The second direction is to try to prove that some two-matroid intersection problem is log-space complete for P. If this could be proved, then it would be very unlikely that the problem can be solved in logarithmic time, although it does not rule out the possibility of other sub-linear time complexities such as $O(\sqrt{n})$ time. It has been proved [GSS-82] that the max-flow problem is log-space complete for P, and the max-flow problem is closely related to the matching problem. Hence, we may get some insight from this proof to prove similar results for the matching problem. This kind of proof is actually a special case of the more general problem of deriving lower bounds for problems in a parallel environment. Deriving lower bounds on parallel computation is currently an active research area.

There are other topics in this thesis which would be interesting to investigate further. For example, we have used the independence oracle and the circuit oracle in the general matroid algorithms. It would be interesting to see how these oracles behave for special cases of two-matroid intersection problems. In order words, how efficiently we can determine independence or circuits in special matroids?

Another question is how the algorithms in this thesis perform on other parallel models, and, conversely, how existing parallel algorithms perform using our proposed models.

The work in this thesis shows that we are at a stage where parallel algorithm design depends on the parallel models, but also gives feedback on possible modifications of the models.

We have proposed modifications to the fetch-and-add CRCW PRAM with the addition of the "write-max" and "write-min" operations. These two operations are very useful in problems where weights are involved. In particular, there have been some investigations of how to find the maximum of a set of numbers in parallel (e.g. [SV-81], [Va-75]), and our new operations suggest a new point of view. We have also introduced a new parallel model called the *concurrent-critical section model* based on the design of parallel algorithms in this thesis. There are many possible ways to design a parallel machine and the measure of how good a machine model is should be how well it can be used to solve problems. Research in this area will provide guidance for the design of real parallel machines in the future. In particular, it would be interesting to know whether it is cost-justifiable to build a machine based on the concurrent-critical section model, and hence achieve O(log$n$) time for resolving critical sections.

APPENDIX : RESULTS

In the following, let m be the number of elements in the matroids, $R \leqslant m$ be the minimum of the ranks of the two matroids, $c1(m)$ and $c2(m)$ be the time complexity for testing dependency in the matroids, and c(m) be max($c1(m)$, $c2(m)$). p is the number of processors used. The sequential times are for the fastest known sequential algorithms. The sequential algorithms for problems (1) and (2) are from [La-76] but the time analysis is improved in this thesis.

(1) The Cardinality Two-Matroid Intersection Problem:

Sequential time [La-76] = $O(mR^2c(m))$

Parallel time:

(i) $O(R(\log R + c(m)))$ if $p \geqslant m^3/\log R$ and using "write-min";

(ii) $O(R(R + c(m)))$ if $p \geqslant \min(mc(m), mR)$ and not using "write-max"/"write-min";

(iii) $O(\frac{mR^2c(m)}{p})$ if $p \leqslant \min(mc(m), mR)$ and not using "write-max"/"write-min"

(2) The Weighted Two-Matroid Intersection Problem:

Sequential time [La-76] = $O(mR^2c(m) + mR^3)$

Parallel time:

(i) $O(R(\log R + c(m)))$ if $p \geqslant m^3$ and using "write-max" and "write-min";

(ii) $O(R(R + c(m)))$ if $p \geqslant mR$ and using "write-max";

(iii) $O(R^2\log m + Rc(m))$ if $p \geqslant mR$ and not using "write-max"/"write-min";

(iv) $O(\frac{mR^2c(m) + mR^3}{p})$ if $p \leqslant mR$ and using "write-max";

(v) $O(\frac{mR^2c(m) + mR^3\log m}{p})$ if $p \leqslant mR$ and not using "write-max"/"write-min"

In the following, let m be the number of arcs and n be the number of vertices in the given graph.

(3) The Unweighted Bipartite Matching Problem:

Sequential time [HK-73] = $O(\sqrt{n}\,m)$

Parallel time:

(i) $O(\sqrt{n}\log n)$ if $p \geqslant n^3/\log n$ and using "write-max";

(ii) $O(n\sqrt{n})$ if $p \geqslant \frac{m}{n}$ and not using "write-max";

(iii) $O(\frac{n^2\sqrt{n}}{p})$ if $p \leqslant \frac{m}{n}$ and not using "write-max"/"write-min"

(4) **The Weighted Bipartite Matching Problem**:

Sequential time [Ta-83] = $O(nm \log_{2+m/n} n)$

Parallel time:

(i) $O(n\log n)$ if $p \geqslant n^3$ and using "write-max" and "write-min";

(ii) $O(n^2)$ if $p \geqslant \frac{m}{n}$ and using "write-min";

(iii) $O(n^2\log n)$ if $p \geqslant \frac{m}{n}$ and not using "write-max"/"write-min";

(iv) $O(\frac{mn}{p})$ if $p \leqslant \frac{m}{n}$ and using "write-min";

(iv) $O(\frac{mn\log n}{p})$ if $p \leqslant \frac{m}{n}$ and not using "write-max"/"write-min";

(5) **Directed Spanning Tree**:

Sequential time [Ta-77] = $O(\min\{m\log n, n^2\})$

Parallel time:

(i) $O(n)$ if $p \geqslant n$ and using "write-max";

(ii) $O(n\log n)$ if $p \geqslant n$ and not using "write-max"/"write-min";

(iii) $O(\frac{n^2}{p})$ if $p \leqslant n$ and using "write-max";

(iv) $O(\frac{n^2\log n}{p})$ if $p \leqslant n$ and not using "write-max"/"write-min"

REFERENCES

[Ar-75] E. Arjomandi. "A Study of Parallelism in Graph Theory." *Ph.D. Dissertation, Dept. of Computer Science*, University of Toronto, 1975.

[AS-83] B. Awerbuch, Y. Shiloach. "New Connectivity and MSF Algorithms for Ultracomputer and PRAM." *Proc. 1983 International Conference on Parallel Processing*, Aug. 1983, pp. 175-179.

[BGH-82] Allan Borodin, Joachim von zur Gathen, and John Hopcroft. "Fast Parallel Matrix and GCD Computation." *Proc. 23rd Symposium on Foundations of Computer Science*, 1982, pp. 65-71.

[BH-82] A. Borodin and J.E. Hopcroft. "Routing, Merging and Sorting on Parallel Models of Computation." *Proc. 14th Symposium on Theory of Computing* (May 1982), pp. 338-344.

[CD-82] S. A. Cook and C. Dwork. "Bounds on the Time for the Parallel RAM's to Compute Simple Functions." *Proc. 14th ACM Symposium on Theory of Computing*, 1982, pp. 231-233.

[CL-65] Yoeng.jin Chu and Tseng.hong Liu. "On the Shortest Arborescence of a Directed Graph." *Scientia Sinica [Peking]* 4(1965). Math. Rev. 33, # 1245, pp. 1396-1400.

[CLC-82] F.Y. Chin, J. Lam and I. Chen. "Efficient Parallel Algorithms for some Graph Problems." *Comm. ACM 25 : 9*, (1982), pp. 659-665.

[Co-83] Stephen A. Cook. "The Classification of Problems which have Fast Parallel Algorithms." *Technical Report, No. 164/83, Dept. of Computer Science, University of Toronto.*

[DPL-80] N. Deo, C.Y. Pang and P.E. Lord. "Two Parallel Algorithms for Shortest Path Problems." *Proc. 1980 International Conference on Parallel Processing*. pp. 244-253.

[DLR-79] D. Dobkin, R.J. Lipton, and S. Reiss. "Linear Programming is Log-space Hard for P". *Information Processing Letters*. 2 (1979), pp. 96-97.

[DNS-81] E. Dekel, D. Nassimi and S. Sahni. "Parallel Matrix and Graph Algorithms." *SIAM J. Comput. 10:4* (Nov. 1981), pp. 657-675.

[DS-82] E. Dekel & S. Sahni. "A Parallel Matching Algorithm for Convex Bipartite Graphs." *Proc. 1982 International Conference on Parallel Processing*, pp. 178-184.

[Ed-68] J. Edmonds. "Optimal Branchings", in *Mathematics and the Decision Sciences*, Part 1, G. Dantzig and A. F. Veinott, Jr., editors, *Amer. Math. Soc. Lectures Appl. Math.* 11 (1968), pp. 335-345.

[EK-72] J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". *Jour. ACM*, 19 (1972), pp. 248-264.

[Fr-81] Andras Frank. "A Weighted Matroid Intersection Algorithm." *Journal of Algorithms*, 2 (1981), pp. 328-336.

[FW-78] S. Fortune and J. Wyllie. "Parallelism in Random Access Machines". *Proc. 10th Symposium on Theory of Computing*, 1978, pp. 114-118.

[GJ-79] Michael R. Garey and David S. Johnson. "Computers and Intractability : A Guide to the Theory of NP-Completeness." *W.H. Freeman and Company*, 1979.

[GK-84] Allan Gottlieb and Clyde P. Kruskal. "Complexity Results for Permuting Data and other Computations on Parallel Processors." *Jour. ACM, 31 : 2*, (April 1984), pp. 193-209.

[Gl-67] F. Glover. "Maximum Matching in a Convex Bipartite Graph." *Naval Res. Logist. Q.*, 14(1976), pp. 313-316.

[Go-78] L.M. Goldschlager. "A Unified Approach to Models of Synchronous Parallel Machines." *Proc. 13th Symposium on Theory of Computing*, 1978, pp. 89-94.

[GP-83] Zvi Galil and Wolfgang J. Paul. "An Efficient General Purpose Parallel Computer." *Jour. ACM, 30 : 2*(1983), pp. 360-381.

[GSS-82] Leslie M. Goldschlager, Ralph A. Shaw, and John Staples. "The Maximum Flow Problem is Log Space Complete for P." *Theoretical Computer Science*, 21(1982), pp. 105-111.

[HK-81] D. Hausmann and B. Korte. "Algorithmic Versus Axiomatic Definitions of Matroids." *Mathematical Programming Study 14*, 1981, pp. 98-111.

[HK-73] John E. Hopcroft and Richard M. Karp. "An $n^{2.5}$ Algorithm for Maximum Matching in Bipartite Graphs." *SIAM J. Comput. 2 : 4*, (Dec 1973), pp.225-231.

[HV-84] D.S. Hirschberg and D.J. Volper. "A Parallel Solution for the Minimum Spanning Tree Problem." *Preliminary draft.*

[Ka-71] R.M. Karp. "A Simple Derivation of Edmonds' Algorithm for Optimum Branchings", *Networks 1* (1971), pp. 265-272.

[Ka-74] A. V. Karzanov. "Determining the Maximal Flow in a Network by the Method of Preflow." *Soviet Math. Dokl.* 15 (1974), pp. 434-437.

[KR-84] S.C. Kwan and W.L. Ruzzo. "Adaptive Parallel Algorithms for Finding Minimum Spanning Trees." *Proc. 1984 International Conference on Parallel Processing*

[Ku-80] H. T. Kung. "The Structure of Parallel Algorithms." *Advances in Computers*, Vol 19, M.C. Yovits, Ed., Academic Press, New York, 1980, pp. 65-112.

[Ku-82] L. Kucera. "Parallel Computation and Conflicts in Memory Access." *Information Processing Letters 14 : 2* (20 Apr. 1980), pp. 93-96.

[La-75] D. Lawrie. "Access and Alignment of Data in an Array Processor." *IEEE Trans. on Computers*,, Vol. C-24, (1975), pp. 1145-1155.

[La-76] Eugene L. Lawler. "Combinational Optimization : Networks and Matroids." *Holt, Rinehart and Winston*, New York, 1976.

[Le-84] T. Leighton. "Tight Bounds on the Complexity of Parallel Sorting." *Proc. 16th Symposium on Theory of Computing*, May 1984, pp. 71-80.

[LD-72] K.N. Levitt and W.T. Kautz. "Cellular Arrays for the Solution of Graph Problems."

Comm. ACM 15 : 9 (Sept. 1972), pp. 789-801.

[LPV-81] Lev, G., N. Pippenger, and L.G. Valiant. "A Fast Parallel Algorithm for Routing in Permutation Networks," *IEEE Trans. on Computers*, 1981.

[MD-81] P. Mateti and N. Deo. "Parallel Algorithms for the Single Source Shortest Path Problem." *Tech. Rep. CS-81-078, Computer Science Dept.,Washington State Univ.*, 1981.

[NYU-83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. "The NYU Ultracomputer : Designing an MIMD Shared Memory Parallel Computer." *IEEE Trans. on Computers, Vol C-32* (Feb 1983), pp. 175-189.

[Pr-83] C.C. Price. "Task Assignment using a VLSI Shortest Path Algorithm." *Tech. Rep., Dept. of Computer Science, Stephen F. Austin State Univ.*, 1983.

[PR-84] Joseph Peters and Larry Rudolph. "Parallel Approximation Schemes for Subset Sum and Knapsack Problems." *Tech. Rep., CMU-CS-84-155, Dept. of Computer Science, Carnegie-Mellon University*, Aug. 1984.

[PS-82] Christos H. Papadimitriou and Kenneth Steiglitz. "Combinatorial Optimization : Algorithms and Complexity." *Prentice Hall, Englewood Cliff*, 1982.

[PV-79] Preparata, F.P., and J. Vuillemin. "The Cube-Connected Cycles." *Proc. 20th Symposium on Foundations of Computer Science*, 1979, pp. 140-147.

[QD-84] Michael J. Quinn and Narsingh Deo. "Parallel Graph Algorithms." *Computing Surveys* 16 : 3, (Sept. 1984), pp. 319-348.

[Qu-83] M.J. Quinn. "The Design and Analysis of Algorithms and Data Structures for the Efficient Solution of Graph Theoretic Problems on MIMD Computers." *Ph.D. dissertion, Computer Science Dept., Washington State Univ.*, 1983.

[Ru-83] Larry Rudolph. "A Robust Sorting Network." *Tech. Rep., Dept of Computer Science, Carnegie-Mellon University*, Aug. 1983.

[Sa-77] C. Savage. "Parallel Algorithms for Graph Theoretic Problems." *Ph.D. dissertation, Mathematics Dept., Univ. of Illinois*, 1977.

[SV1-82] Yossi Shiloach and Uzi Vishkin. "An $O(n^2 \log n)$ Parallel MAX-FLOW Algorithm." *Journal of Algorithms 3*, (1982), pp. 128-146.

[SV2-82] Yossi Shiloach and Uzi Vishkin. "An O(logn) Parallel Connectivity Algorithm." *Journal of Algorithms*, 3(1982), pp. 57-67.

[SV-81] Yossi Shiloach and Uzi Vishkin. "Finding the Maximum, Merging and Sorting in a Parallel Computation Model." *Journal of Algorithms 2*, (1981), pp. 88-102.

[Ta-77] R.E. Tarjan. "Finding Optimum Branching." *Networks 7*, (1977), pp. 25-35.

[Ta-83] Robert Endre Tarjan. "Data Structures and Network Algorithms." *CBMS-NSF Regional Conference Series in Applied Mathematics 44, SIAM*, 1983.

[Ta-85] Eva Tardos. "A Strongly Polynomial Minimum Cost Circulation Algorithm." *Combinatorica*, May 1985.

[Va-75] L.G. Valiant. "Parallelism in Comparison Problems." *SIAM J. Comput. 4 : 3* (1975), pp. 348-355.

[Vi-83] Uzi Vishkin. "Synchronous Parallel Computation - A Survey." *Courant Institute, New York University*, April 1983.

[We-76] D.J.A. Welsh. "Matroid Theory." *Academic Press*, New York, 1976.

[Yo-83] Y.B. Yoo. "Parallel Processing for some Network Optimization Problems." *Ph.D. Dissertion, Computer Science Dept., Washington State Univ.*, 1983.

[Za-84] Vasilii Zakharov. "Parallelism and Array Processing." *IEEE Trans. on Computers, Vol C-33*, (Jan. 1984), pp. 45-78.