Fast Parallel Algorithms for Graphs and Networks

by

Danny Soroker

Fast Parallel Algorithms for Graphs and Networks

By

Danny Soroker

B.Sc. (Technion Israel Institute of Technology) 1981
M.Sc. (Technion Israel Institute of Technology) 1983

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: Richard M Karp 11/20/87
Chairman Date

Manuel Blum 11/9/87

Dorit Hochbaum 11/16/87.

.....

Fast Parallel Algorithms For Graphs and Networks

Danny Soroker

ABSTRACT

Many theorems in graph theory give simple characterizations for testing the existence of objects with certain properties, which can be translated into fast parallel algorithms. However, transforming these tests into algorithms for *constructing* such objects is often a real challenge. In this thesis we develop fast parallel ("*NC*") algorithms for several such construction problems.

The first part is about tournaments. (A tournament is a digraph in which there is precisely one arc between every two vertices.) Two classical results state that every tournament has a Hamiltonian path and every strongly connected tournament has a Hamiltonian cycle. We derive efficient parallel algorithms for finding these objects. Our algorithms yield new proofs of these theorems. In solving the cycle problem we also solve the problem of finding a Hamiltonian path with one fixed endpoint. Next we address the problem of constructing a tournament with a specified degree-sequence, and give an *NC* algorithm for it which achieves optimal speedup.

The second part is concerned with making graphs strongly connected via orientation and augmentation. A graph is strongly orientable if its edges can be assigned orientations to yield a strongly connected digraph. Robbins' theorem states the conditions for a graph to be strongly orientable. His theorem was generalized for mixed graphs, i.e. ones that have both directed and undirected edges. We give a fast parallel algorithm for strongly orienting mixed graphs. We then solve the problem of adding a minimum number of arcs to a mixed graph to make

it strongly orientable. This problem was not previously known to have even a polynomial time sequential solution (a sequential algorithm was discovered independently by Gusfield). In the process of solving the general problem we derive solutions for the special cases of undirected and directed graphs.

The final part of the thesis describes a methodology which yields deterministic parallel algorithms for several supply-demand problems on networks with zero-one capacities. These problems include: constructing a zero-one matrix with specified row- and column-sums, constructing a simple digraph with specified in- and out-degrees and constructing a zero-one flow pattern in a complete network, where each vertex has a specified net supply or demand. We extend our results to the case where the input represents upper bounds on supplies and lower bounds on demands.

This research was supported by Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871, Monitored by Space & Naval Warfare Systems Command under Contract No. N00039-84-C-0089 and by the International Computer Science Institute, Berkeley, California.

Richard M. Karp

Chairman of the Committee: Richard M. Karp

To Edeet and Tamar - my family !

Acknowledgements

It is a pleasure to acknowledge those who contributed to the creation of this document.

First and foremost I would like to thank my advisor Dick Karp. I feel fortunate to have known him and worked with him. His suggestions of research problems initiated most of the work in this dissertation. His quickness in detecting errors and his many insights and suggestions were crucial in keeping me on the right track and molding my ideas into working solutions. Dick's friendliness and sense of humor made him very approachable. He made time to meet me and listen to my half baked ideas just a few hours before flying to the Far East for a month. He has been inspiring as a teacher and researcher.

I thank Manuel Blum and Dorit Hochbaum for serving on my committee. It was always fun discussing research and life with Manuel. His originality and uniqueness shine. His warmth glows. I enjoyed my conversations with Dorit. I thank her for her careful reading of the manuscript and her valuable comments.

Noam Nisan and I worked jointly on chapter five of this thesis. He was an excellent collaborator, always coming up with new ideas. Working with him was a very enjoyable experience, as was learning to sail together. I enjoyed working with Phil Gibbons. His diligence, ideas and eye for detail were instrumental in getting results. My first collaborator in Berkeley was Howard Karloff (our joint work appears in his thesis). He was fun to work with and I thank him for introducing me to the world of Ultimate Frisbee.

Berkeley is a great place for doing theory. I thank David Shmoys for helping me realize this and for being a main force in "luring" me into the Theory group. Gene Lawler's friendliness also helped. Many visitors contributed to the exciting atmosphere, of whom I would like to acknowledge Amos Fiat, Mike Sipser, Vijaya Ramachandran and Avi Wigderson. Special thanks go to Vijaya, whom I had the pleasure of working with, for expressing interest in my research.

One of the best parts of "The Berkeley Experience" has been to get to know and live amongst my fellow students in the Theory group. It was wonderful to

have Sampath Kannan next door. Talking with him about research and just about anything else was always enjoyable, even after loosing the nth game of badminton to him. Valerie King and Joel Friedman showed me the wonders of cross-country skiing in Yosemite. Steven Rudich was always surprising. I am grateful to Marshall Bern, Jon Frankle, Lisa Helerstein, Moni Naor, Prabhakar Ragde, Umesh Vazirani, Alice Wong, Yanjun Zhang and the rest of the Theory crowd for making my life richer.

Last and most I thank my wife, Edeet. Her love and support kept me going. Our daughter, Tamar, was born just in time to be mentioned here, and was kind enough to come with me to the office several times to help me type in the last few pages of this dissertation.

Table of Contents

CHAPTER ONE

INTRODUCTION

Parallel computation is becoming a central field of research in computer science. The main driving forces behind this have been technological advances in the area of Very Large Scale Integration. The price of computer hardware has been pushed down to the point where time of execution (rather than cost of components) is the main limiting factor in many applications. Furthermore, the speed of processor and memory units has been constantly increasing, but is gradually approaching saturation due to physical limitations. The combination of these factors has brought forth the necessity for parallel computers - machines which have many processors that cooperate and coordinate actions efficiently to solve problems quickly.

The theory of parallel computation is still in early stages of development, but is gaining popularity due to its practical importance on one hand and to the fundamental mathematical problems arising in it on the other. The body of knowledge on parallel algorithms is much smaller than that on sequential algorithms. One reason is, of course, *time* - researchers have been constructing sequential algorithms ever since the idea of "computer" was born, whereas development of a coherent theory of parallel computation started only in the late 1970s. Thus, the body of knowledge on sequential complexity of problems is much larger than that on parallel complexity.

Another main difference lies in deciding on an appropriate model of computation. In the sequential world there is a clear resemblance between the standard Random Access Machine model ([AHU]) and actual working computers - algorithms developed for this model can be translated quite simply into programs in existing languages. This is not the case in parallel computing. The prevailing theoretical model for algorithm design is the Parallel Random Access Machine (PRAM, to be formally defined in chapter two). It has an unbounded number of processors working in full synchrony, each performing its own instruction at any given step and each having unit-time access to a shared memory. It is set apart

from current machines in several aspects:

(1) MIMD vs. SIMD - In the PRAM model different processors perform different operations at any given step. This property is known as MIMD (Multiple Data Multiple Instruction). However, machines with the largest number of processors currently available, such as the Connection Machine of Thinking Machines Corp. which has as many as 65,536 processors ([Hillis]), are SIMD (Single Instruction Multiple Data). This means that at each time step all processors perform the same operation. Current MIMD machines, such as the NYU Ultracomputer ([GGKMRS]) and the BBN Butterfly ([RT]), have substantially fewer processors (several hundred).

(2) Shared vs. distributed memory - Both the Ultracomputer and the Butterfly resemble the PRAM model in that they provide shared memory. However access to the shared memory takes more time than local operations. Furthermore, a large part of the design effort has gone into constructing the interconnection network between processors and the shared memory, and scaling these designs to accommodate for larger numbers of processors seems to be a very complex and expensive task. One solution to this problem is to have a network of processors with no shared memory, in which processors communicate by sending messages. One such design is the Caltech Cosmic Cube upon which some commercial computers, such as the Intel iPSC, are based ([SASLMW]). Several strong theoretical results ([Upfal],[KU],[Ranade]) show that PRAMs can be efficiently simulated by bounded degree networks (i.e. ones in which the degree of a processor (the number of neighbors it has) is fixed, independent of the total number of processors). These results give partial justification for using the PRAM model for algorithm design.

(3) Synchronous vs. asynchronous - The PRAM is a highly synchronous model. Each step of the computation is performed synchronously by all processors. This is impractical under current technology because of problems in achieving clock synchronization. However, some form of synchronization is important to have and some existing machines provide tools for this. For example, the NYU Ultracomputer has a "replace-add" operation built into its hardware, which can be used for synchronization.

Finally we note that in some of the currently most powerful "supercomputers", such as the Cray-2 and the Goodyear Aerospace Corp. MPP ([SASLMW]) most of the parallelism is in the form of pipelining and vector operations.

To summarize, the PRAM model of computation differs substantially from existing parallel computers. Even so, it is an appropriate model to use when studying the limits and possibilities of parallelism. It frees the algorithm designer from worrying about details which are not a fundamental part of the problem being studied. Furthermore, as stated above, it is possible to automatically translate PRAM programs to ones on more realistic models (bounded degree networks) with a relatively small loss of efficiency. Finally, the PRAM model sets a goal for future designers of parallel computers.

This thesis is a study in parallel algorithms. The research reported here involved developing parallel algorithms for several basic problems related to graphs and networks. Graph theory is a fundamental area underlying computer algorithm design, since graphs are general structures, which provide a convenient means for modeling many real-world problems. Our motivation for choosing the problems we chose was not necessarily because they arise in specific applications, but rather that the known sequential algorithms for them seem hard to parallelize.

It is interesting to point out two features that are shared by most of the problems we considered, since they often come up in the study of parallel algorithms:

(1) The problem has a simple, even trivial, sequential algorithm. Transforming this algorithm into a fast parallel algorithm turns out to be a big challenge, and often a totally new approach is needed.

(2) The problem is a search or construction problem (e.g "find a set with property X") which has a related decision problem ("does there exist a set with property X?"). The decision problem has an known fast parallel solution whereas the search problem does not. This seemingly inherent difference between decision and search problems does not come up usually in polynomial time sequential computation. This is because much of the research in sequential algorithms has been on search problems. Furthermore, in many cases there is an obvious means of converting an algorithm for a decision problem to an algorithm for the related search problem (using self reducibility). However,

this conversion seems inherently sequential and does not yield a fast parallel algorithm. An interesting discussion appears in [KUW1].

There were several goals motivating the research reported here. First, as mentioned above, the problems considered posed a challenge in that their known sequential solutions seem hard to parallelize. Second, in the process of trying to create parallel algorithms for problems, *basic techniques* for parallel computation may be developed. By "basic techniques" we mean procedures that have potentially wide applicability in helping to solve many problems. Examples of such techniques in the literature are parallel "tree contraction" of Miller and Reif ([MR]), the "Euler Tour" technique for trees of Tarjan and Vishkin ([TV]) and "independence systems" of Karp, Upfal and Wigderson ([KUW1]). Examples of techniques developed in this thesis are: introducing the notion of a "dense matching" with an efficient parallel implementation for it (chapter four) and the idea of partitioning the edges of a graph into "constellations" (chapter five).

Another possible benefit from devising parallel algorithms is that new insight can be obtained for sequential computation. The reason for this is that trying to find a parallel solution to a problem seem to require, in many cases, directions which are very different than the common sequential ones. On the other hand, a parallel algorithm (in the model we will be using) can easily be converted into a sequential one. Therefore, a new efficient parallel algorithm (where efficiency is measured by the total number of operations performed by the algorithm) yields a new sequential algorithm with good running time. For example, motivations from parallel complexity led us to solve the minimum strong augmentation problem for mixed graphs (chapter four), for which no previous polynomial-time sequential algorithm was published. A sequential algorithm was discovered independently by Gusfield ([Gusf]), who gives several applications of this problem.

The process of developing parallel algorithms can be broken down into two main stages, analogous to the sequential case. First (at least in the problems we considered) one needs to determine if the problem on hand has a fast parallel solution. Here we need to define what "fast parallel" means. We will be using the (by now standard) notion of *NC* (to be formally defined in chapter two) for this purpose. This first stage can be very hard, and is analogous to showing that a prob-

lem is in P. It is possible to give evidence that a problem is unlikely to be in NC by showing it to be "P-complete" (see chapter two), analogously to marking a problem as "probably hard" by proving it to be NP-complete. The second stage is to find ways to improve the efficiency of the algorithm found in the first stage. This is analogous to developing sophisticated data structures to push down the running time of sequential algorithms as close as possible to optimal. In this context we will define the notions of "efficient" and "optimal" parallel algorithms.

We now give a brief outline of the thesis. Chapter two contains a formal description of the concepts which are the foundations of the research reported here. We define our model of computation, relevant complexity classes and efficiency of algorithms. We also mention some basic graph theoretic terminology and notation we will be using.

Chapter three deals with tournaments, which are digraphs in which each pair of vertices is connected by one arc. The first half of this chapter has algorithms for constructing Hamiltonian paths and cycles in tournaments ([Soro1]). The second half describes a method for constructing a tournament with a given degree sequence ([Soro3]). This problem is similar to the problems considered in chapter five, but the techniques for solving it are very different.

Chapter four describes solutions to several problems related to edge orientation ([Soro2]). The first problem is to orient the edges of a mixed graph (i.e. a graph with some directed and some undirected edges) so that the resulting digraph is strongly connected. A graph for which such an orientation is possible is called strongly orientable. Next we solve the problem of augmenting a graph (or digraph) with a minimum number of edges to make it strongly orientable. Finally we extend these methods to derive a solution for the minimum strong augmentation problem for mixed graphs.

Chapter five contains a description of a methodology which yields fast parallel algorithms for several construction problems that can be stated as supply-demand problems with zero-one arc capacities ([NS]). The problems solved are constructing a zero-one matrix with specified row- and column-sums, constructing a digraph with given in- and out-degree sequences and constructing a zero-one flow pattern between a set of sites, each of which has a specified supply or demand. The algo-

rithms are extended to the case where the input specifies upper bounds on supplies and lower bounds on demands.

CHAPTER TWO

FUNDAMENTAL THEORETICAL CONCEPTS

In this chapter we give a formal discussion of the main concepts relevant to this thesis. We will discuss the PRAM model of computation, the complexity classes *NC* and *RNC* and efficient and optimal algorithms. Finally, we will mention the basic graph theoretic terminology and notation we will be using.

2.1 The PRAM Model of Computation

Many models of parallel computation have appeared in the literature (a good survey appears in [Cook1]). In this manuscript we will be focusing on one model: the Parallel Random Access Machine (PRAM). This model was defined by various researchers (see e.g. [Vish1] for a survey), and is currently the prevailing model for designing parallel algorithms. A PRAM contains a sequence of processors, $P_1, P_2, P_3, \ldots$ and a shared memory. Each processor has a local memory and the capabilities of the standard sequential random-access machine ([AHU]): it can, in one step, perform a basic operation such as adding two numbers, comparing two numbers, reading or writing to its local memory etc. A processor can also access the shared memory in the manner described below. Every processor has a unique integer index distinguishing it from all other processors.

The PRAM is **synchronous** - the computation is performed in parallel steps, where at step i each processor performs its ith computation step. In each parallel step of computation every processor performs one of three primitive operations: it can either read one cell of the shared memory, write into one cell of the shared memory or perform a local computation. If, in one step, some cell is both read from and written into, we will assume that the read occurs before the write. PRAMs are categorized according to the allowable configurations in accessing the shared memory.

Exclusive/Concurrent Read: in an exclusive read PRAM, no two processors attempt to read the same memory cell at the same step. An algorithm is valid

for this model only if it has this property. In a concurrent read machine any number of processors can read a memory cell in one step.

Exclusive/Concurrent Write: an exclusive write PRAM is one in which no two processors attempt to read the same memory cell at the same step. In a concurrent write model many processors can try to write into the same cell simultaneously (i.e. in the same step). Here (as opposed to concurrent read), it is not obvious what the result of a concurrent write should be, so a further categorization is necessary:

COMMON: all processors attempting to write simultaneously into the same cell must write the same value.

ARBITRARY: one of the processors attempting to write simultaneously in one cell succeeds (i.e. its value gets written). The algorithm may not make any assumptions about which of the processors succeeds.

PRIORITY: the processor with the lowest index of those attempting to write simultaneously in one cell succeeds.

The type of PRAM is specified mnemonically, for example CREW means concurrent read - exclusive write. It is clear that concurrent read (write) is at least as powerful as exclusive read (write), since any algorithm for one can also run on the other. Similarly, PRIORITY is the most powerful concurrent write model (of the ones listed above) and COMMON is the weakest.

All the processors have the same program, but at a given time different processors will be performing different operations since the program can refer to the processors' identification numbers. The requirement of one program for all processors makes the model more practical and also disallows unreasonable power, in that it enforces a 'uniformity' condition (i.e. the same program works for different input sizes) and implies that the program is finite. For example, if each processor had a different program, there would exist a PRAM program that solves the halting problem: the program of processor i would be: " if the input is i and Turing machine i halts then output 'halts'; otherwise output 'doesn't halt' ".

A PRAM computation works as follows: initially the input appears in the first n cells of shared memory. The first $p(n)$ processors are simultaneously "activated", and run a common program as described above, where the function, $p(n)$, depends

on the program. We will refer to this function as "the *number of processors* used by the algorithm. It is assumed that each processor knows the value n. At the end of the computation the output should appear at a specified location (or set of locations) in the shared memory. The number of steps until the last active processor finishes running the program is the *running time* of the algorithm.

There is an obvious time-processor tradeoff which is important to point out: let $c>1$ be a real number and say a certain PRAM program, A, runs in time t and uses p processors. Then A can be modified to use only $\lceil \frac{p}{c} \rceil$ processors, and run in time $O(ct)$. The modification is simply to have each processor in the modified program do the work of c processors in A. Therefore each step of A corresponds to c consecutive steps in the modified program. As a special case of this, A yields a sequential algorithm running in time $p \cdot t$.

A note about word size. We will assume that a memory cell (shared or local) is of size $c\log_2 n$ (for some constant, c, depending on the algorithm). Consequently, basic operations (addition, comparison) on integers up to n^c can be performed in one step. One reason for making this assumption is that we will be dealing with graphs, and usually the primitive elements will be names of vertices and edges.

Randomization: As in sequential computation, the notion of *randomized* algorithms helps in solving certain problems. To this end we define a *probabilistic PRAM*. In this model each processor has the additional capability of generating random numbers. More specifically, we will assume that in one step a processor can generate a random word, i.e. an integer chosen uniformly in the range $[0,n^c]$. Again, a probabilistic PRAM can be either exclusive or concurrent read and write.

2.2 Complexity Classes

In our study of parallel algorithms we need to define appropriate complexity classes to express the notion of a "fast parallel algorithm". The class NC is commonly used for this purpose. The class NC was first identified and characterized by Pippenger ([Pipp]). The name "NC" was coined by Cook (see e.g. [Cook1]) and stands for "Nick's Class", giving credit to Nick Pippenger.

Definition: A decision problem is in NC if there exists a PRAM algorithm solving it that runs in time $O(\log^{c_1} n)$ using $O(n^{c_2})$ processors, where n is the size of the input and c_1 and c_2 are constants independent of n.

First we note that NC is defined for decision problems (i.e. problems for which the output is one bit). We will use the term more loosely and talk about an *NC algorithm*, i.e. a PRAM algorithm that obeys the appropriate time and processor constraints.

The next thing to note is that the type of PRAM is not specified in the definition. The reason is that the strongest PRAM model (PRIORITY CRCW) can be efficiently simulated by the weakest model (EREW). More precisely, for any p and t, for any PRIORITY CRCW algorithm that runs in time t and uses p processors, there exists an equivalent EREW algorithm that runs in time $O(t \cdot \log p)$ and uses p processors. (Equivalent means having the same input-output correspondence.) The simulation (due to Vishkin) involves sorting all the accesses to the shared memory, thus detecting the highest-priority processor writing into each cell and providing for concurrent writes. Since sorting can be done in time $O(\log n)$ using a linear number of processors on an EREW PRAM ([Cole]), the simulation achieves the complexity stated above. It is standard to make a finer classification within NC:

Definition: A decision problem is in AC^k ($k \geq 1$) if there exists a PRIORITY CRCW PRAM algorithm solving it that runs in time $O(\log^k n)$ using $O(n^c)$ processors, where n is the size of the input and c is a constant independent of n. A problem is in AC if it is in AC^k, for some $k \geq 1$.

It is clear by the statements above that $AC = NC$. We point out that in the literature NC is commonly defined in terms of another computational model, uniform circuit families. When defining NC using this model, one gets a finer classification within NC into classes NC^k (along the lines of the classes AC^k within AC). We will not elaborate on this here, since the definitions given above are sufficient for our purposes. For more details regarding other models and classes related to NC see [Cook1] and [Cook2].

It is clear by the definition (and by a previous remark about time-processor tradeoff) that NC is contained in P, the class of problems solvable in (sequential) polynomial time. It is generally believed to be *strictly* contained, but researchers are currently very far from proving general lower bounds that would imply this separation. However, because of this belief, there is a way of "giving strong evidence" that a problem in P is not in NC, by showing that its membership in NC would imply $P=NC$. Such a problem is *log−space complete for P* or simply *P−complete.*

A *log−space reduction* is a transformation between problems computable by a Turing machine with logarithmic work space (details in [GJ]). A problem, X, is P-complete if $X \in P$ and every problem in P is log-space reducible to X. As in the theory of NP-completeness, one shows that a problem is P-complete by demonstrating a (log-space) reduction from a problem that is known to be P-complete to it. Quite a few problems have been shown to be P-complete. The **circuit value problem** (given the description of a boolean circuit and values for its inputs, what is the value of the output?) was shown to be P-complete by a "generic reduction" (a reduction from *any* problem in P given a Turing machine solving it in time bounded by some polynomial in the input length), and plays a similar role to SAT in the theory of NP-completeness ([Ladner]). Other interesting examples of P-complete problems are max flow ([GSS]) and finding the lexicographically first maximal clique ([Cook2]).

How is a log-space reduction relevant for parallel computation? It turns out that there is a close relationship between sequential space and parallel time known as the "parallel computation thesis" (see e.g. [FW]). A consequence of this is that a log-space reduction can be computed in NC. Thus if a P-complete problem is in NC then every problem in P is in NC. It is, therefore, unlikely that a P-complete problem is in NC.

The probabilistic counterpart of NC is *Random NC* (RNC).

Definition: A decision problem is in RNC if there is a probabilistic PRAM algorithm that runs in polylog time using a polynomial number of processors which, on any input, gives the correct answer with probability at least $\frac{3}{4}$ ([Cook2]).

The definition can be extended to problems with many output bits by saying that the algorithm gives a correct output sequence with probability at least $\frac{3}{4}$. An algorithm of the type appearing in the definition (i.e. one that can make errors) is known as a *Monte Carlo* algorithm. A more powerful notion is a probabilistic algorithm that makes no errors, known as a *Las Vegas* algorithm. In this case the random variable is the running time of the algorithm, whose expected value is (for our purposes) polylog in the input size.

2.3 Efficient and Optimal Algorithms

We have identified the notion of problems having fast parallel algorithms with the class NC. However, it is not clear that an NC algorithm would be practical, since the number of processors in an actual machine does not change as a function of the input size. For example, say the best NC algorithm we have for some problem uses n^3 processors and the same problem has a sequential algorithm that runs in time n. If we have an actual machine with, say, 1000 processors then for instances larger than 31 the sequential algorithm would run faster using one processor than the parallel algorithm using all 1000 processors (implementing the processor-time tradeoff mentioned earlier). We, therefore, need a more precise notion of what constitutes a good parallel algorithm.

Definition: Let $T(n)$ be the running time of the fastest sequential algorithm known for solving problem X, and let A be a PRAM algorithm solving X that runs in time $t(n)$ and uses $p(n)$ processors. Then:

(1) A is *efficient* if $p(n)\cdot t(n)=O(T(n)\cdot\log^c n)$ (for some constant, c).

(2) A is *optimal* (or *achieves optimal speedup*) if $p(n)\cdot t(n)=O(T(n))$.

The first thing to note is that this definition is quite pragmatic - it evaluates an algorithm according to the current state of knowledge (i.e. the best sequential algorithm known), which is not necessarily well-defined theoretically, but makes sense practically. The definition is theoretically sound for problems which are known to have matching upper and lower sequential bounds.

Another point regarding this widely used definition is that an algorithm that achieves optimal speedup does not necessarily have a high degree of parallelism. For example, the best *sequential* algorithm is optimal by the above definition. Therefore a more meaningful notion for our purposes is an *efficient (optimal) NC* algorithm (i.e. an *NC* algorithm that is efficient or optimal).

Some basic problems that have efficient *NC* algorithms are: computing connected components of an undirected graph ([SV]), computing biconnected components ([TV]), constructing a maximal matching in a graph ([IS]) and dynamic expression evaluation ([MR]). Problems for which optimal *NC* algorithms are known include sorting ([Cole]) and prefix computations (e.g. [Fich]).

2.4 Terminology and Notation

The graph-theoretic terminology we use is mostly standard. We give here some of the main definitions (to fix terminology rather than to introduce concepts). For a wider discussion see [CL] or [Berge]. More specialized definitions (e.g. for tournaments and mixed graphs) will be given in the appropriate chapters.

An **(undirected) graph** $G=(V,E)$ consists of a set of vertices, V, and a set of edges, E. $V(G)$ and $E(G)$ denote, respectively, the vertex set and edge set of G. An **edge**, $e=\{u,v\}$, is an unordered pair of vertices (u and v are the **endpoints** of e, and e is **incident** to u and v). The **degree** of a vertex is the number of edges incident to it. Two vertices, u and v, are **adjacent** (or **neighbors**) if $\{u,v\}\in E$. A **path** is a sequence of edges $\{v_1,v_2\}, \{v_2,v_3\}, \ldots, \{v_{k-1},v_k\}$ where $v_i\neq v_j$ for all $i\neq j$. If $v_1=v_k$ then this sequence of edges is a **cycle**. A path can also be expressed by the sequence of vertices along it. A **forest** is a graph with no cycles. Vertices x and y are in the same **connected component** if they lie on some path. G is **connected** if all the vertices are in the same connected component. A **tree** is a connected graph with no cycles. A **bipartite graph** is a graph whose vertex set is partitioned into two sets, X and Y, such that every edge is incident to a vertex in X and a vertex in Y.

A **directed graph** (or **digraph**), $G=(V,E)$, consists of a set of vertices, V, and a set of arcs, E. $V(G)$ and $E(G)$ denote, respectively, the vertex set and arc set

of G. An **arc**, $a=(u,v)$, is an ordered pair of vertices (u is the **tail** of a and v is its **head**). The **in-degree** $d_{in}(v)$ (**out-degree** $d_{out}(v)$) of v is the number of arcs whose head (tail) it is. A **(directed) path from** v_1 **to** v_k is a sequence of arcs $(v_1,v_2), (v_2,v_3), \ldots, (v_{k-1},v_k)$ where $v_i \neq v_j$ for all $i \neq j$. If $v_1=v_k$ then this sequence is a **(directed) cycle**. Vertices x and y are in the same **strongly connected component** if there is a path from x to y and from y to x. G is **strongly connected** if all the vertices are in the same strongly connected component.

The following definitions are appropriate for both graphs and digraphs (with "edge" representing both edges and arcs). The number of vertices and edges will usually be denoted by n and m respectively. A **subgraph** of G is a graph, H, whose vertex set is a a subset of $V(G)$ and edge set is a subset of $E(G)$. H is a **spanning subgraph** of G if $V(H)=V(G)$. A **Hamiltonian path (cycle)** is a spanning path (cycle). For a set of vertices, U, the **induced subgraph on** U (denoted by $G(U)$) is the graph on the vertex set U whose edge set is $\{\{x,y\} \mid x,y \in U \text{ and } \{x,y\} \in E(G)\}$.

CHAPTER THREE

EFFICIENT ALGORITHMS FOR TOURNAMENTS

3.1 Introduction

A *tournament* is a directed graph $T=(V,E)$ in which any pair of vertices is connected by exactly one arc. This models a competition involving n players, where every player competes against every other one. A trivial but useful fact is that any induced subgraph of a tournament is also a tournament. If $(u,v) \in E$ we will say that *u dominates v* , and denote this property by $u > v$. Note that since the directions of the arcs are arbitrary, the domination relation is not necessarily transitive. We extend the notion of domination to sets of vertices: let A,B be subsets of V. *A dominates B* $(A > B)$ if every vertex in A dominates every vertex in B.

For a given vertex, v, we categorize the rest of the vertices according to their relation with v : $W(v)$ is the set of vertices that are dominated by v (i.e. vertices involved in matches which v *Won*) and $L(v)$ is the set of vertices that dominate v (matches which v *Lost*).

The *transitive tournament* on n vertices is the tournament in which each integer between 1 and n has a corresponding vertex, and i dominates j if $i > j$. The *score* of a vertex is the number of vertices it dominates. The *score list* of a tournament is the sorted list of scores of its vertices (starting with the lowest).

Tournaments have been extensively studied (e.g. [BW], [Moon]). In this chapter we will deal with several classical results. The first set of results talk about existence of Hamiltonian paths and cycles in tournaments: every tournament has a Hamiltonian path, and every strongly connected tournament has a Hamiltonian cycle. These theorems are in contrast with the fact that deciding if an arbitrary graph is Hamiltonian is *NP*-complete [GJ]. The proofs of these theorems in the literature imply efficient algorithms for finding these objects, but since the proofs are by induction, the algorithms seem inherently sequential. A natural question is - can Hamiltonian paths and cycles in tournaments be found quickly in

parallel? We answer in the affirmative by giving *NC* algorithms for both problems. In the process of giving the algorithms we demonstrate new proofs of the theorems.

At first we show how to find a Hamiltonian path. A similar algorithm was discovered independently by J. Naor ([Naor]). Our solution uses an interesting technical lemma, which states that in every tournament there is a "mediocre" player - one that has both lost and won many matches. We also give a very simple and efficient randomized algorithm, which raises some interesting issues.

We then move to the Hamiltonian cycle problem, which turns out to be quite a bit more complicated. The main idea in the solution is defining a new problem - that of finding a Hamiltonian path with one fixed endpoint - and solving it simultaneously with finding a Hamiltonian cycle, using a "cut and paste" technique.

The other main part of this chapter deals with the construction problem: given a non-decreasing list of integers, $\vec{s}=s_1,\ldots,s_n$, determine if there exists a tournament with score list $\vec{s}$, and if so, construct such a tournament. A simple, non-constructive criterion for testing if such a list is a score list was found by Landau in 1953 ([BW]): $\vec{s}$ is a score list of some tournament if and only if, for all k, $1\leq k\leq n$:

$$\sum_{i=1}^{k} s_i \geq \binom{k}{2}$$

with equality for $k=n$.

A simple greedy algorithm ([BW,CL]) is known for constructing a tournament with v_i having score s_i (for all $1\leq i\leq n$): select some score s_i and remove it from the list; have v_i dominate the s_i vertices with smallest scores (and have the rest of the vertices dominate v_i); subtract 1 from the score of each vertex dominating v_i and repeat this procedure for the reduced list. We note that very similar algorithms exist for several other construction problems, some of which are discussed in chapter 5 of this thesis ([Berge],[CL],[FF]).

Checking the set of conditions described above is easy to do efficiently in parallel, but implementing the construction algorithm seems hard. We give an alternate method, which yields an optimal *NC* algorithm for the construction problem.

The algorithms presented in this chapter are efficient, some optimal. All the deterministic algorithms use $O(n^2/\log n)$ processors on a CREW PRAM, where n is the number of vertices. They are efficient since the size of the tournament is $\Theta(n^2)$. The algorithms for Hamiltonian path and cycle run in time $O(\log^2 n)$. The algorithm for the tournament construction problem runs in time $O(\log n)$.

The randomized algorithm for the Hamiltonian path problem runs in expected time $O(\log n)$ on a CRCW PRAM **and uses only $O(n)$ processors!** At first sight this seems "better than optimal", but it is observed that only $\Theta(n \log n)$ arcs need to be inspected in order to find a Hamiltonian path. However, for all the other problems considered in this chapter, $\Omega(n^2)$ lower bounds hold: in finding strongly connected components and Hamiltonian cycles $\Omega(n^2)$ arcs need to be inspected (as can be shown by a simple adversary strategy); in the tournament construction problem the output size is $\Theta(n^2)$. Thus the algorithms described for these problems are, indeed, efficient or optimal.

We start with a discussion of the structure of the strongly connected components of a tournament, which will be useful in later sections. This structure has some nice properties which give rise to an optimal NC algorithm for finding the strongly connected components. In contrast, the most efficient parallel algorithm known for determining strongly connected components in general digraphs is to compute the transitive closure, and is not optimal. This is discussed in more detail in the introduction to chapter 4.

3.2 Strongly Connected Components of a Tournament

An important computation required by our algorithms is finding strongly connected components in a tournament. In this section we show some special properties of the strong component structure of tournaments and give a simple optimal NC algorithm for finding it based on these properties. **In a nutshell, the strong component structure of a tournament depends only on its score list!**

The first simple lemma shows that there is a total ordering of the strongly connected components.

Lemma 3.1: Let T be a tournament and let $C_1, C_2, \cdots, C_k$ be its strongly con-

nected components. Then for all i, j either $C_i > C_j$ or $C_j > C_i$ (recall that $A > B$ means that every vertex in A dominates every vertex in B).

Proof: By definition of strongly connected components all arcs between C_i and C_j go in the same direction. Since T is a tournament all such arcs exist. □

The implication of this lemma is that in order to describe the strong component structure we need only specify the partition of vertices into strongly connected components and the total order of the components (as opposed to a general digraph for which there is only a partial ordering of the strongly connected components). The next lemma shows that this partition is related to the score list.

Lemma 3.2: Let $u, v \in V$ and say $s(u) \geq s(v)$ (where $s(u)$ is the score of vertex u). Let S_u and S_v be the strongly connected components containing u and v respectively. Then either $S_u = S_v$ or $S_u > S_v$.

Proof: If $u > v$ the claim clearly holds. If $v > u$, then by the pigeonhole principle there must be a vertex, w, such that $u > w$ and $w > v$. Thus the claim holds in this case too. □

Let $V = \{v_1, \ldots, v_n\}$, where $s(v_1) \leq s(v_2) \leq \cdots \leq s(v_n)$. Lemma 3 tells us that each strongly connected component is of the form $v_j, v_{j+1}, \ldots, v_k$, for some j, k. How can we determine where a strongly connected component starts and where it ends? This turns out to be simple: If v_k is the vertex of highest score in a strongly connected component, then $v_i < v_j$ for all $i \leq k < j$. It follows that $\sum_{i=1}^{k} s(v_i)$ - the number of arcs whose tail is in the set $\{v_1, \ldots, v_k\}$ - is equal to $\binom{k}{2}$ - the number of arcs in the tournament induced on this vertex set. The converse is also true: if $\sum_{i=1}^{k} s(v_i) > \binom{k}{2}$, then v_k and v_{k+1} are in the same strongly connected component. Thus the strongly connected components can be computed by the following algorithm:

procedure *STRONG_COMPONENTS*(T)

(1) In parallel for all vertices, v, compute the score, $s(v)$, of v.

(2) Sort the sequence of scores in non-decreasing order to obtain the score list,

$\vec{s} = s_1, s_2, \cdots, s_n$ of T.

(3) Compute the partial sums, $p_k = \sum_{i=1}^{k} s_i - \binom{k}{2}$ for all $1 \le k \le n$ in parallel.

(4) Partition the vertices into strongly connected components according to the zeroes in the sequence $p_1, p_2, \ldots, p_n$: the vertex with score s_k is the last (i.e. of highest score) vertex in a strongly connected component if and only if $p_k = 0$.

end *STRONG_COMPONENTS*.

It is not hard to see that this procedure can be performed using $O(n^2/\log n)$ processors in time $O(\log n)$ on a CREW PRAM. In fact, if the scores are given then only $O(n)$ processors are required (using Cole's sorting algorithm for step (2) [Cole]).

To summerize, the structure of the strongly connected components of a tournament depends only on its score list. This may seem surprising, since $\Theta(n^2)$ bits are required to specify a tournament on n vertices, but only $O(n \log n)$ bits are needed to specify its score list.

3.3 Hamiltonian Paths and Cycles

3.3.1 Hamiltonian Path

We start by stating the theorem due to Redei [Redei] and its textbook proof ([Rober] page 487).

Theorem 3.1: Every tournament contains a Hamiltonian path.

Proof: By induction on the number, n, of vertices. The result is clear for $n = 2$. Assume it holds for tournaments on n vertices. Consider a tournament, T, on $n+1$ vertices. Let v be an arbitrary vertex of $V(T)$. By assumption $G(V - \{v\})$ has a Hamiltonian path $v_1, v_2, \ldots, v_n$. If $v > v_1$ then $v, v_1, \ldots, v_n$ is a Hamiltonian path of T. If $v < v_n$ then $v_1, \ldots, v_n, v$ is a Hamiltonian path of T. Otherwise there must exist an index, $i < n$, such that $v_i > v$ and $v_{i+1} < v$. In this case, $v_1, \ldots, v_i, v, v_{i+1}, \ldots, v_n$ is the desired Hamiltonian path. □

This proof yields a very efficient sequential algorithm for finding a Hamiltonian path in a tournament. The important observation is that adding a new vertex to a path of length l can be done by inspecting only $O(\log l)$ arcs using binary search. This shows that only $O(n\log n)$ arcs need to be inspected in the worst case in order to find a Hamiltonian path. A matching lower bound can be obtained by an analogy to *sorting*: when the given tournament is transitive, finding a Hamiltonian path in it is equivalent to sorting n integers, since inspecting an arc corresponds to a comparison, and the Hamiltonian path corresponds to the sorted sequence. Since $\Omega(n\log n)$ comparisons are required for sorting ([AHU]), it follows that $\Omega(n\log n)$ arcs need to be inspected to determine a Hamiltonian path in a tournament.

In order to obtain a fast parallel algorithm a different method seems to be required. The approach we take is divide and conquer. A simple-minded way is the following: (i) Split the tournament into two subgraphs, T_1, T_2, of roughly equal order. (ii) In parallel, find Hamiltonian paths H_1 in T_1 and H_2 in T_2. (iii) Connect H_1 and H_2 to form a Hamiltonian path of T.
The problem with this approach is that step (iii) is not guaranteed to succeed, since we have no control over what the endpoints of H_1 and H_2 are.

It turns out that a slightly modified approach does work. The key observation is the following: let v be a vertex of T. Consider Hamiltonian paths $l_1, \ldots, l_k$ of $L(v)$ and $w_1, \ldots, w_m$ of $W(v)$. Since $l_k > v$ and $v > w_1$ we can obtain the following Hamiltonian path of T: $l_1, \ldots, l_k, v, w_1, \ldots, w_l$. Note that this provides an alternative, simpler proof of theorem 1.

In order to derive an NC algorithm from the above idea, we need the following technical lemma:

Lemma 3.3 (Mediocre player lemma): In a tournament, T, on n vertices there exists a vertex, v, for which both $L(v)$ and $W(v)$ have at least $\lfloor \frac{n}{4} \rfloor$ vertices.

Proof: We point out that this lemma is a corollary of lemma 3.4. The proof given here is of interest because it shows why the constant $\frac{1}{4}$ comes up and implies what the worst-case tournaments are. Let

$$I = \{u \mid d_{in}(u) \geq d_{out}(u)\}$$

$$O = V - I.$$

Assume w.l.o.g that $|I| \geq |O|$. By the pigeonhole principle there exists a vertex, v, whose out-degree in $T(I)$ is no less that its in-degree in $T(I)$. Thus

$$d_{out}(v) \geq \lfloor \frac{|I|}{2} \rfloor \geq \lfloor \frac{n}{4} \rfloor$$

and $d_{in}(v) \geq d_{out}(v) \geq \lfloor \frac{n}{4} \rfloor$ by definition. []

Remark: A simple construction shows that for every n there are tournaments on n vertices for which each vertex has either in-degree or out-degree $\lfloor n/4 \rfloor$.

Using lemma 1 we obtain our algorithm:

procedure $PATH(T)$

(1) Let n = order of T.

(2) If $n = 1$ then return the unique vertex of T.

(3) Find a vertex, v, whose in-degree and out-degree in T are both at least $\lfloor n/4 \rfloor$.

(4) In parallel find $H_1 = PATH(L(v))$ and $H_2 = PATH(W(v))$.

(5) Return the path (H_1, v, H_2).

end $PATH$.

By lemma 1, step (3) can be achieved, so only $O(\log n)$ levels of recursive calls (step (4)) are required. The time required for one level is $O(\log n)$ on a CREW PRAM (partitioning the vertices and updating their degrees). Therefore the total running time is $O(\log^2 n)$. The number of processors required is $O(n^2/\log n)$ (since the degree of a vertex can be computed in $O(\log n)$ time with $O(n/\log n)$ processors by standard methods).

The main obstacle in making this algorithm more efficient is the need to compute the in- and out-degrees of the vertices and to update them in every step. In fact it can be shown that any deterministic algorithm for finding a "mediocre" vertex in a tournament needs to inspect $\Omega(n^2)$ arcs in the worst case. We do not know, at this point, if there is a deterministic NC algorithm for the Hamiltonian path problem that uses only $O(n)$ processors, but there is a simple *randomized*

scheme based on the observation that "most vertices are mediocre". This is formally stated in the following lemma, which is a generalization of lemma 3.3:

Lemma 3.4: In a tournament, T, on n vertices there are at least $n-4k+2$ vertices for which $|L(v)| \geq k$ and $|W(v)| \geq k$, for all $1 \leq k \leq \lfloor \frac{n}{4} \rfloor$.

Proof: Let S_k be the set of vertices whose out-degree is less than k. The cardinality of S_k is no more than $2k-1$, since the tournament induced on S_k contains a vertex whose out-degree is at least $\lfloor \frac{|S_k|}{2} \rfloor$. Similarly, there are at most $2k-1$ vertices whose in-degree is less than k. Therefore, at least $n-4k+2$ vertices have both in- and out-degrees at least k. []

Lemma 3.4 says, for example, that at least half the vertices in a tournament have both in- and out-degrees at least $n/8$. This implies that the following randomized algorithm will be very efficient:

procedure $R\_PATH(T)$

(1) If T has one vertex then return the unique vertex of T.

(2) Select a random vertex, v, of T.

(3) In parallel find $H_1 = R\_PATH(L(v))$ and $H_2 = R\_PATH(W(v))$.

(4) Return the path (H_1, v, H_2).

end $PATH$.

Lemma 3.5: The expected depth of recursion of R_PATH is $O(\log n)$.
Furthermore,

$$For\ all\ c \geq 10 \qquad Pr[\ recursion\ depth\ > c\log_{8/7} n\ ] \ < \ n^{-c}$$

Proof: Let us say that a given stage of the algorithm, whose input is a subtournament on some number, s, of vertices, is <u>successful</u> if, for the vertex chosen in step (2), both $W(v)$ and $L(v)$ have no more than $7s/8$ vertices. Let x be some vertex. We can describe the history of x throughout the algorithm by a zero-one vector, V_x, where $V_x[i]=1$ if and only if the i'th stage in which x participated in was successful. By definition, V_x contains at most $\log_{8/7} n$ 1's. As is remarked above, the probability of success of any stage is at least 1/2. Therefore,

$Pr[V_x[i]=1] \geq 1/2$ independently for all i. From standard results on tails of the binomial distribution we get:

$$For\ all\ c \geq 10 \quad Pr[\ length\ of\ V_x > c\log_{8/7} n] < n^{-(c+1)}$$

Now, since there are n vertices, the probability that some vector, V, is long is at most n times the probability that V_x is long, which yields the statement of the lemma. The statement about the expected depth of recursion is a simple consequence. []

It turns out that repeatedly selecting a random vertex in each of the sub-tournaments that are created is a nontrivial matter (if one wants to do it in expected constant time). The difficulty arises from the fact that a vertex knows which sub-tournament, S, it belongs to at any given stage, but it does not know which other vertices belong to S. We will not discuss this here, only state that it can be done. Thus our randomized algorithm works almost surely in time $O(\log n)$ and uses $O(n)$ processors (one for each vertex) on a probabilistic CRCW PRAM, and is, therefore, optimal.

3.3.2 Hamiltonian Cycle and Restricted Path

The following theorem, due to Camion [Camion] (see [BW] page 173), states exactly when a tournament has a Hamiltonian cycle:

Theorem 3.2: A tournament is Hamiltonian if and only if it is strongly connected.

The "only if" part is trivial. The other direction is proven by induction, but a similar proof to that of theorem 3.1 will not work here, since removal of a vertex from a strongly connected tournament might result in a tournament which is not strongly connected. A classical proof, due to Moon [Moon] (see [BW] page 173), proves a stronger claim: a strongly connected tournament on n vertices has a cycle of length k, for $k=3,4,\ldots,n$. We omit the proof.

Again, the proof yields an efficient algorithm, which seems sequential in nature. For our parallel solution we introduce a new notion - a restricted Hamiltonian path.

Definition: A *restricted Hamiltonian path* is a Hamiltonian path with a specified

endpoint (either the first or the last vertex, not both).

A natural question is - when does there exist a Hamiltonian path starting (ending) at a given vertex, v? The next theorem gives the precise condition.

Definition: Let T be a tournament and v be a vertex in T. v is a *source* (*sink*) of T if all vertices of T have directed paths from (to) v.

Theorem 3.3: A tournament, T, has a Hamiltonian path starting (ending) at vertex v if and only if v is a source (sink) of T.

Proof: Again, the "only if" part is trivial. We prove the second direction of the theorem for a source. The proof is symmetrical for a sink. The proof is by induction on the n, the order of T. For $n=1$ the claim holds trivially. Assume the claim for tournaments of n vertices. Let T be a tournament of order $n+1$, and let v be a source of T. Using the inductive claim we need only show that $W(v)$ contains a source of $G(V-\{v\})$. By theorem 3.1, $W(v)$ contains a Hamiltonian path starting at, say, u. Thus u is a source of $W(v)$. Furthermore, by assumption every vertex in $L(v)$ can be reached from some vertex in $W(v)$. Thus u is a source of $G(V-\{v\})$. □

Once again the proof implies a sequential algorithm. The key idea for an NC algorithm for finding a Hamiltonian cycle in a strongly connected tournament is to tie it to the problem of finding restricted Hamiltonian paths. The idea is that each problem will be solved by recursive calls to the other. We start by giving an alternative proof for theorem 3.3, this time using theorem 3.2.

Second Proof of theorem 3.3: Let $C_1, C_2, \cdots, C_k$ be the strongly connected components of T such that $C_1 > C_2 > \cdots > C_k$. Since v is a source of T, it must lie in C_1. Since C_1 is strongly connected, it contains a Hamiltonian cycle, H_1. Let H_2 be the path obtained by deleting from H_1 the unique arc entering v. We note that H_2 is a Hamiltonian path of C_1 starting at v. Let H_3 be a Hamiltonian path of $\{C_2, C_3, \ldots, C_k\}$. By construction, the last vertex of H_2 dominates the first vertex of H_3, so (H_2, H_3) is a Hamiltonian path of T starting at v. □

Now we return to theorem 3.2 and prove it using theorem 3.3.

New Proof of theorem 3.2: Let T be a strongly connected tournament and let

$v \in V(T)$. Let $L_1 > L_2 > \cdots > L_q$ be the strongly connected components of $L(v)$ and $W_1 < W_2 < \cdots < W_p$ be the strongly connected components of $W(v)$. Since T is strongly connected there must be some arc leaving W_1. Every such arc must go to a vertex in $L(v)$ (Since, by definition, it cannot go to a vertex in W_i, $i > 1$, or to v). Let:

$$m = \min\{i \mid a > b \text{ for some } a \in W_1,\ b \in L_i\},$$

and let $w_1 \in W_1$, $l_1 \in L_m$ be such that $w_1 > l_1$. Symmetrically, there must be an arc entering L_1 and let:

$$k = \min\{i \mid a > b \text{ for some } a \in W_i,\ b \in L_1\},$$

$w_2 \in W_k$, $l_2 \in L_1$ and $w_2 > l_2$. The construction is shown in fig. 3.1.

The existence of a Hamiltonian cycle of T is shown by demonstrating several paths and the connections between their endpoints. These paths are shown in fig. 3.2. The paths are the following:

(1) A Hamiltonian path of W_1 ending at w_1.

(2) A Hamiltonian path of $\{L_m, L_{m+1}, \ldots, L_q\}$ starting at l_1.

(3) The vertex v.

(4) A Hamiltonian path of $\{W_k, W_{k+1}, \ldots, W_p\}$ ending at w_2.

(5) A Hamiltonian path of L_1 starting at l_2.

(6) A Hamiltonian path of $\{W_2, W_3, \ldots, W_{k-1}, L_2, L_3, \ldots, L_{m-1}\}$

We claim that the concatenation of the paths above in the order (1),(2),(3),(4),(5),(6) forms a Hamiltonian cycle of T. First notice that each of the paths specified does, in fact, exist. For the restricted paths ((1), (2), (4) and (5)) this is a consequence of theorem 3.3. The only other fact we need to verify is that the arcs between endpoints of the paths are in the desired direction. The only non-obvious cases are the connections from path (5) to path (6) and from path (6) to path (1). For showing this recall that we chose L_m and W_k in a way that implies that $L_2, L_3, \ldots, L_{m-1}$ all dominate W_1 and $W_2, W_3, \ldots, W_{k-1}$ are all dominated by L_1. Thus the last vertex of path (5) must dominate the first vertex of path (6). Similarly, the last vertex of path (6) must dominate the first vertex of path (1). Notice that both endpoints of path (6) may be either in $L(v)$ or in $W(v)$. □

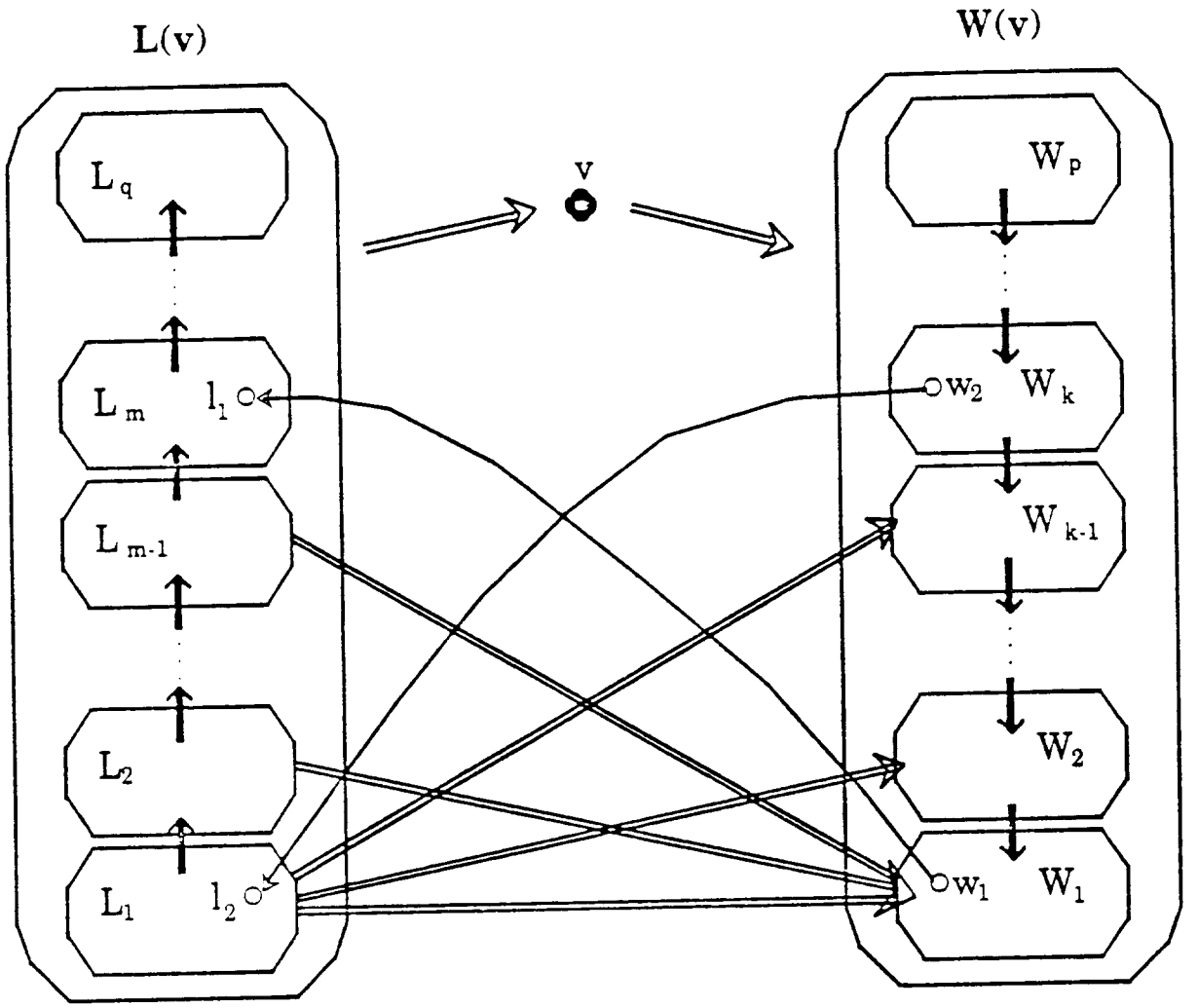

Fig. 3.1: The construction used in the second proof of theorem 3.2.

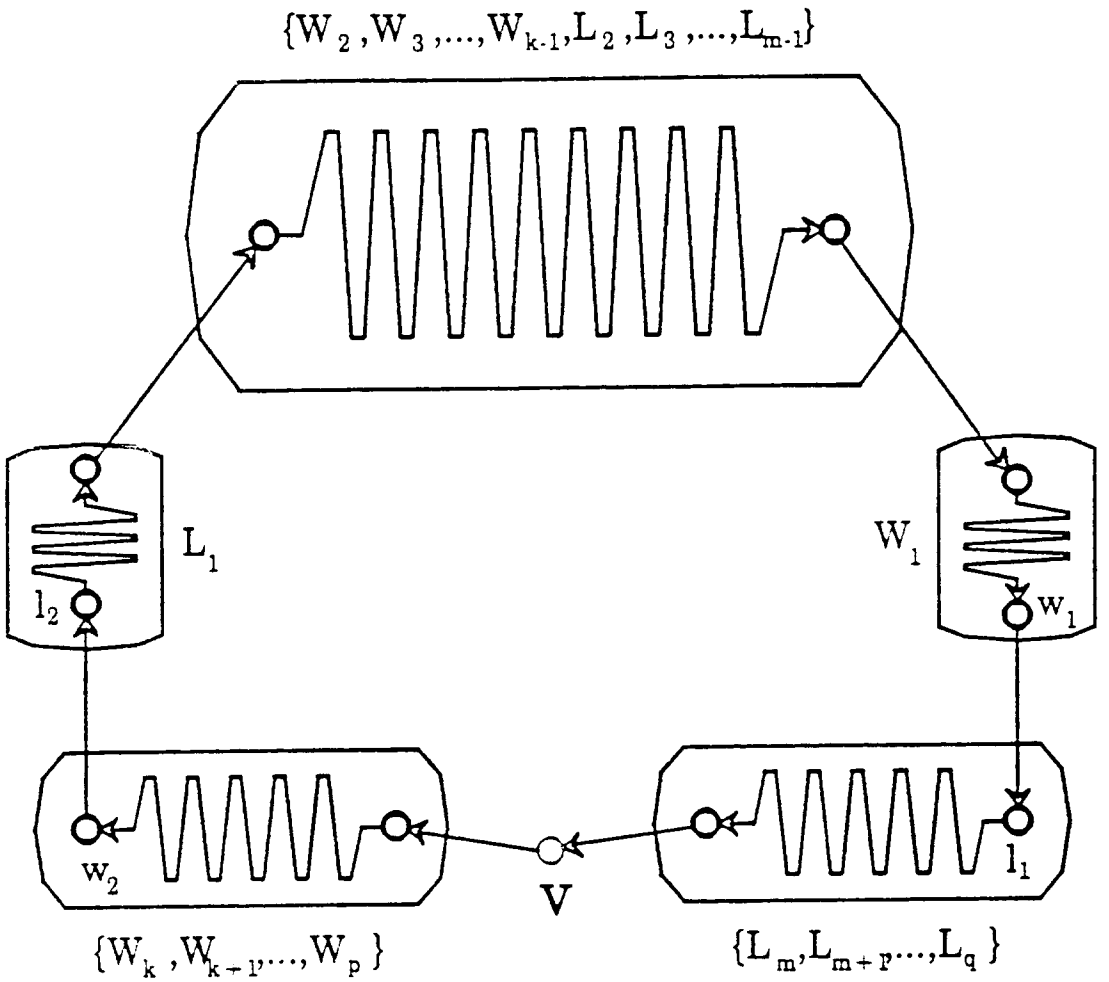

Fig. 3.2: Demonstration of the Hamiltonian cycle in the proof of theorem 3.2.

This new proof gives an approach for an NC algorithm - by selecting v to be a "mediocre" vertex we break the problem into several subproblems of bounded size: subgraphs (1), (2), (3), (4) and (5) all have at most $\frac{3}{4}n$ vertices. However, subgraph (6) (the union of components $W_2, \ldots, W_{k-1}$ and $L_2, \ldots, L_{m-1}$) may be very large. In fact it may contain all but five vertices of T, since v, w_1, w_2, l_1 and l_2 are the only vertices guaranteed to be outside of this subgraph.

It turns out that this apparent obstacle is non-existent! The critical observation is that the Hamiltonian path we need to find in (6) is *not restricted*. Therefore we can use procedure *PATH* for finding this path, and need not worry about the size of this subproblem. Thus the problem of finding a Hamiltonian cycle (or restricted Hamiltonian path) on n vertices breaks down into several similar problems, each on no more than $\frac{3}{4}n$ vertices, and one easier problem on at most n vertices.

The algorithms for Hamiltonian cycle and restricted path follow. Note that the solution to the Hamiltonian cycle problem is very symmetrical, as demonstrated in fig. 3.2.

procedure *RESTRICTED_PATH*(T,*endpoint*,u)

(1) Let n = order of T.

(2) If $n=1$ then return the unique vertex of T.

(3) Find strongly connected components $C_1 > C_2 > \cdots > C_k$ of T.

(4) If endpoint='start' then

(4.1) In parallel find

$H_1 = CYCLE(C_1)$

$H_2 = PATH(\{C_2, \ldots, C_k\})$.

(4.2) Let $H_1 = H_1 - \{$*unique* arc *into* $u\}$.

(5) If endpoint='end' then

(5.1) In parallel find

$H_1 = PATH(\{C_1, \ldots, C_{k-1}\})$

$H_2 = CYCLE(C_k)$

(5.2) Let $H_2 = H_2 - \{unique$ arc $out\ of\ u\}$.

(6) Return the path (H_1, H_2).

end *RESTRICTED_PATH*.

procedure *CYCLE*(T)

(1) Let n = order of T.

(2) If $n = 1$ then return the unique vertex of T.

(3) Find a vertex, $v \in T$, whose in-degree and out-degree in T are both at least $\lfloor n/4 \rfloor$.

(4) Find strongly connected components $L_1 > \ldots > L_q$ of $L(v)$ and $W_1 < \ldots < W_p$ of $W(v)$.

(5) In parallel find

$$m = \min\{i \mid a > b \text{ for some } a \in W_1,\ b \in L_i\},$$

$$k = \min\{i \mid a > b \text{ for some } a \in W_i,\ b \in L_1\},$$

and $w_1 \in W_1$, $l_1 \in L_m$, $w_2 \in W_k$, $l_2 \in L_1$ such that $w_1 > l_1$ and $w_2 > l_2$.

(6) In parallel find

$H_1 = RESTRICTED\_PATH(W_1, 'end', w_1)$

$H_2 = RESTRICTED\_PATH(\{L_m, \ldots, L_q\}, 'start', l_1)$

$H_3 = RESTRICTED\_PATH(\{W_k, \ldots, W_p\}, 'end', w_2)$

$H_4 = RESTRICTED\_PATH(L_1, 'start', l_2)$

$H_5 = PATH(\{W_2, \ldots, W_{k-1}, L_2, \ldots, L_{m-1}\})$

(7) Return the cycle $(v, H_3, H_4, H_5, H_1, H_2, v)$

end *CYCLE*.

We now indicate how these algorithms can be performed using $O(n^2/\log n)$ processors in $O(\log^2)$ time on a CREW PRAM. Finding strongly connected components can be done using these resources, as described in section 3.2. Finding the minimum-index component that has an arc in a given direction to another component can be done by a standard prefix computation on a subset of the arcs of T (e.g. [Fich]). Finally, in each stage we need to compute *PATH*. This seems to be a problem since *PATH* itself takes $O(\log^2 n)$ time and the recursion depth is $O(\log n)$.

The observation here is that the result returned from *PATH* is not required in order to generate the recursive calls to *CYCLE* and *RESTRICTED_PATH*. Therefore all the calls to *PATH* can be performed separately from the main recursion (for example after completing it), and then all the paths (restricted and non-restricted) can be connected together in the appropriate manner. Since no vertex or arc appears in more than one call to *PATH*, this additional step (of all calls to *PATH*) can be done with the stated resources.

3.3.3 Open Problems

We have shown that finding a Hamiltonian path and a restricted Hamiltonian path in a tournament are both in *NC*. A natural question is: what is the complexity of finding a *doubly restricted Hamiltonian path* in a tournament, T, i.e a Hamiltonian path from a specified vertex, a, to another specified vertex, b. We know how to solve this problem in *NC* if either of the graphs T, $T-\{a\}$, $T-\{b\}$ or $T-\{a,b\}$ is not strongly connected. However, if all these graphs are strongly connected, we do not even know if the problem is solvable in polynomial time.

Another interesting problem is whether there is an efficient deterministic *NC* algorithm for finding a Hamiltonian path in a tournament. As stated above, the sequential complexity of this problem is $\Theta(n\log n)$, whereas the complexity of finding a "mediocre" vertex is $\Theta(n^2)$. Therefore a totally different approach is required to solve the path problem efficiently in parallel. It might be possible to show that no such algorithm exists by proving that any algorithm that asks about n arcs in one step needs many steps (i.e. more than poly-log) in the worst case before it discovers a Hamiltonian path in a tournament.

3.4 The Tournament Construction Problem

3.4.1 The Upset Sequence

Our approach for constructing a specified tournament is based on, what we call, the upset sequence of a tournament, T, which describes the difference between T and a transitive tournament. If we list the vertices according to their scores in non-decreasing order, then an *upset* is when a vertex, v, dominates some

other vertex appearing later than v in the list. We call an arc corresponding to an upset a *reverse arc.* Transitive tournaments are exactly those tournaments that contain no upsets.

Definition: Let $s_1 \leq \cdots \leq s_n$ be the score list of a tournament, T, and let v_i be the vertex of score s_i (for all $1 \leq i \leq n$). The *upset sequence* of T, is the sequence, $\vec{u}$, where u_k is the number of upsets between $\{v_1, \ldots, v_k\}$ and $\{v_{k+1}, \ldots, v_n\}$ (for all $1 \leq k \leq n-1$).

The score list uniquely determines the upset sequence (and vice-versa):

Lemma 3.6: Let T be a tournament with score list $\vec{s}$ and upset sequence $\vec{u}$. Then, for all $0 \leq k \leq n-1$:

$$u_k = \sum_{i=1}^{k}(s_i - i + 1) = \sum_{i=1}^{k} s_i - \binom{k}{2}$$

Proof: There are exactly $\binom{k}{2}$ arcs in the subgraph induced on $\{v_1, \ldots, v_k\}$, since it is also a tournament. Therefore the right hand side describes the number of arcs whose tail is in $\{v_1, \ldots, v_k\}$, but whose head isn't. []

Corollary 3.1: For all $1 \leq k \leq n$:

$$s_k = u_k - u_{k-1} + k - 1$$

How can we use the upset sequence? Our approach is to construct a tournament with a given score list by starting with a transitive tournament and reversing some of its arcs. The upset sequence of the desired tournament gives us a handle on which arcs to reverse. We will be aided by a graphical representation of the upset sequence, which we now discuss.

A sequence of non-negative integers can be represented graphically by its histogram. We will treat the histogram as a rectilinear polygon (and call it, simply, a *polygon*), which is divided into squares, each of which has integral x and y coordinates. The x coordinate is a square's *column* and the y coordinate is its *height.* An example of a polygon is shown in fig. 3.3. Any collection of squares of a polygon is a *sub-polygon.* A maximal set of consecutive squares at the same height is called a *slice.* Note that a polygon can have several slices at the same height (if it is not convex). A (horizontal) *segment* is consecutive set of squares, all in the same slice.

We denote a segment or slice by $[l,r]$ or by $[l,r;h]$, where l and r are, respectively, the columns of the leftmost and rightmost squares it contains, and h is its height.

A polygon representing the upset sequence of a tournament will be called an *upset polygon.*

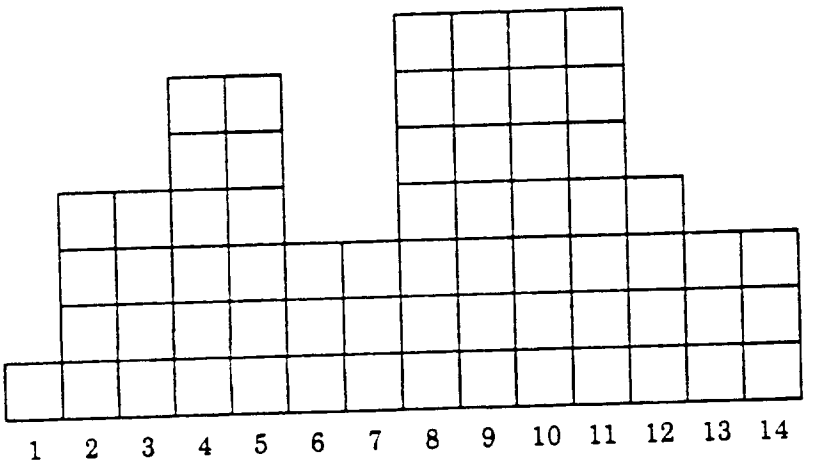

Fig. 3.3: A polygon representing the sequence 1,4,4,6,6,3,3,7,7,7,7,4,3,3.

An elementary property of a polygon, which follows from its definition is:

Proposition 3.1: The slices of a polygon form a nested structure: if $[l_1,r_1]$ and $[l_2,r_2]$ are slices with $l_1 \geq l_2$ then either $l_1 > r_2$ or $r_1 \leq r_2$.

We define the following partitioning problem: Given a rectilinear polygon as shown in fig. 3.3, partition each of its slices into segments such that no two segments in the partition agree on both endpoints. Such a partition is said to be *valid*, and is defined by the set of segments it contains. An example of a valid partition is illustrated in fig. 3.4. The partition is {[1,14], [2,4], [2,5], [2,14], [4,4], [4,5], [5,5], [5,14], [8,8], [8,9], [8,10], [8,11], [9,11], [10,11], [11,12]}.

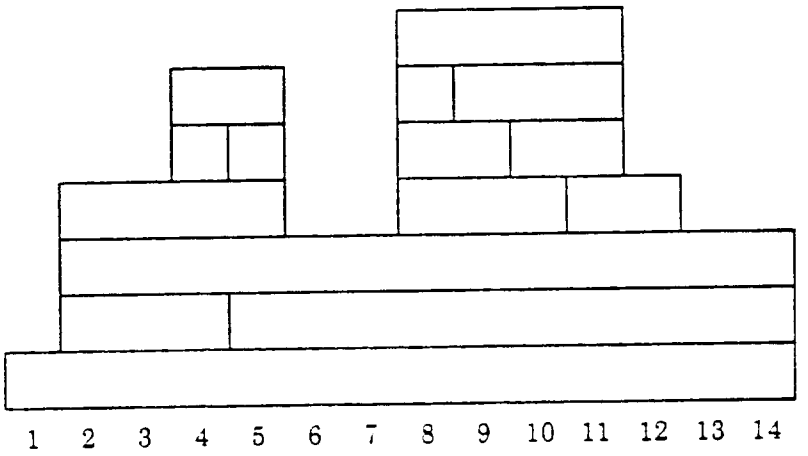

Fig. 3.4: A valid partition of the polygon of fig. 3.3.

Lemma 3.7: A valid partition of the upset polygon yields a solution to the construction problem.

Proof: Let $\{[l_i,r_i] \mid 1 \le i \le m\}$ be the set of segments in a valid partition of an upset polygon representing a sequence $\vec{u}$ corresponding to the score list $\vec{s} = s_1, \ldots, s_n$. Let T be the tournament obtained by taking the n vertex transitive tournament and reversing the arcs $\{(r_i,l_i) \mid 1 \le i \le m\}$. By inspection, the number of reverse arcs crossing the cut $(\{v_1, \ldots, v_k\}:\{v_{k+1}, \ldots, v_n\})$ is exactly u_k. Therefore (by corollary 3.1), T is a tournament with score list $\vec{s}$. □

Note that each slice in fig. 3.4 is partitioned into at most two segments. This is not a coincidence.

Definition: A *2−partition* is a valid partition in which every slice is partitioned into at most 2 segments. A slice which is partitioned into at most 2 segments is *2−partitioned*.

We will deal only with 2-partitions because of the following:

Lemma 3.8: If a polygon has a valid partition, then it has a 2-partition.

Proof: Let P be a valid partition of some polygon, which is not a 2-partition. Let S be a slice which is partitioned into more than 2 segments such that all slices lying above S are 2-partitioned. We will prove the lemma by showing how to transform P into another valid partition in which S is 2-partitioned and the partition of slices above S is unchanged.

Let the segments comprising S in P be, from left to right, $[l_1,r_1], \ldots, [l_k,r_k]$ (where $k>2$). If either $[l_1,r_{k-1}]$ or $[l_2,r_k]$ does not appear in P, then the partition of S can be replaced with $\{[l_1,r_{k-1}],[l_k,r_k]\}$ or $\{[l_1,r_1],[l_2,r_k]\}$ respectively. If both appear, then at least one, say $[l_1,r_{k-1}]$, must appear in a slice below S (call this slice T). This follows from the assumption that all slices lying above S are 2-partitioned and from the nesting property (proposition 3.1). Now, simply assign the segment $[l_1,r_{k-1}]$ to S and the segments $[l_1,r_1], \ldots, [l_k,r_k]$ to T. □

Not every rectilinear polygon of the type discussed has a valid partition. Two examples are shown in fig 3.5.

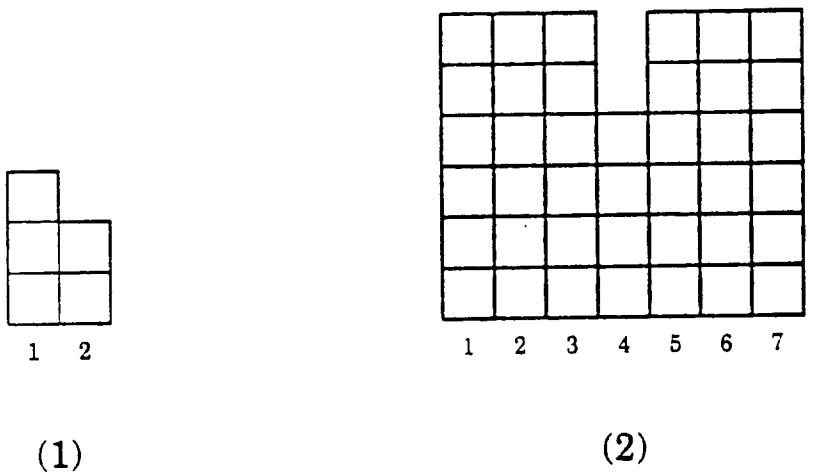

Fig. 3.5: Examples of polygons which have no valid partition.

We will show, however, that every upset polygon has a 2-partition. A few more definitions are required for this: a *left (right) face* is a maximal vertical line segment on the left (right) part of the boundary of a polygon. *Face k*, if it exists, is the face between columns $k-1$ and k. Two faces, L and R, are *opposing* if there is some slice starting at L and ending at R. The *width, w(F)* of a face, F, is the *minimum* distance between it and any of its opposing faces (where distance is measured by number of squares). The length of a face F (i.e the number of slices it touches) is denoted by $l(F)$.

Lemma 3.9: A polygon, D, has a 2-partition if the length of every face of D is no more than half its width.

Proof: We prove the lemma by induction on the height of D. If the height is 1

then D clearly has a 2-partition. Assume the claim holds for all polygons of height $k-1$, and let k be the height of D. Let D' be the polygon obtained by removing the bottom slice from D. By the inductive assumption, D' has a 2-partition, P. We will show that P can be extended to a 2-partition of D. Let L and R be, respectively, the left and right faces bounding the bottom level of D. P contains $l(L)-1$ segments starting at L and $l(R)-1$ segments ending at R. By the condition of the lemma,

$$\textit{width of bottom slice} \geq w(L)\ ,\ w(R) \geq l(L)+l(R)$$

Therefore, by the pigeonhole principle, there are two segments that partition the bottom slice, which are not contained in P. Thus P can be extended to become a 2-partition of D. []

Lemma 3.10: In an upset polygon the length of every face is no more than half its width.

Proof: Let $\Delta(k)$ be the difference in height between the highest square with x-coordinate k and the highest square with x-coordinate $k-1$. In other words, if F is a left face bounding squares with x-coordinate k, then $\Delta(k)=l(F)$. If F is a right face then $\Delta(k)=-l(F)$. Using corollary 3.1:

$$\Delta(k) = u_k - u_{k-1} = s_k - k + 1$$

Since $\vec{s}$ is non-decreasing, it follows that:

$$(*) \quad \textit{for all } 2 \leq k \leq n-1 \qquad \Delta(k) \geq \Delta(k-1)-1$$

Say face k is a left face, L. The nearest opposing face of L occurs to the right of the first value, r, such that $r>k$ and $\Delta_{k+1}+\Delta_{k+2}+\cdots+\Delta_r<0$. The smallest possible r value can occur (by (*)) when $\Delta_k=\Delta_{k+1}+1=\cdots=\Delta_r+r-k$. In this case:

$$w(L) = r-k+1 = 2\Delta(k) = 2l(L)$$

A symmetric argument works for right faces. []

Theorem 3.4: Every upset polygon has a 2-partition.

3.4.2 2-Partitioning the Upset Polygon

As described in the previous section, our algorithm works as follows: given the score list, $\vec{s}$, we compute its corresponding upset sequence $\vec{u}$ and construct a 2-

partition, P, of the upset polygon. In the output tournament, for all $1 \leq i < j \leq n$, v_i dominates v_j if and only if $[i,j] \in P$.

What remains to be shown is how to compute a 2-partition of an upset polygon, U, efficiently in parallel. Basically, our approach is to construct the partition according to faces. We first observe that it is a simple task to partition a set of slices with a common face as follows: say the common face is a left face. Let the set of slices be, from top to bottom, $S_1, \ldots, S_m$, where $S_i = [l, r_i]$ for all $1 \leq i \leq m$. Then S_i will be partitioned into the segments $[l, l+i]$ and $[l+i+1, r_i]$. This is shown in fig. 3.5. Such a partition is always possible given lemma 3.10. A symmetric partition exists for slices sharing a right face.

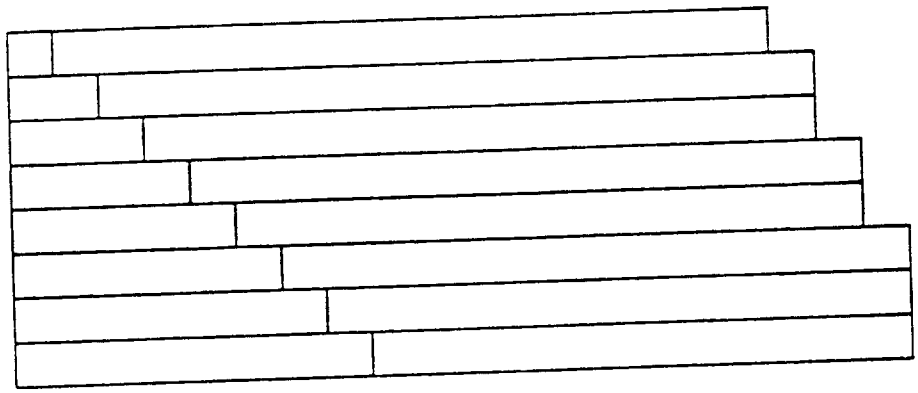

Fig. 3.5: 2-partitioning a set of slices with a common left face.

If we simultaneously partition the entire polygon in the manner described (according to left faces), the resulting partition might not be valid, since a right face can be opposite several left faces. Our solution is to have every slice "belong" to one of (the two) faces it touches, and to be partitioned accordingly. More specifically, it belongs to the dominant face according a domination relationship defined as follows: a left face, L, *dominates* an opposing right face, R, unless the top slice touching L touches R but the top slice touching R *does not* touch L (in other words, R is the highest face opposing L but not vice-versa).

Theorem 3.5: Let $S = [l,r,h]$ be a slice belonging to face F. Let $S_F = [l',r',h']$ be the highest slice belonging to F. Say we partition S into 2 segments such that the length of the segment touching F is $h' - h + 1$. If we perform this partitioning for all the slices of an upset polygon, U, then the result is a (valid) 2-partition of U.

Proof: First we note that if two slices belong to the same face, they must be of

different height, so their partition cannot conflict (i.e. create segments with identical endpoints). Therefore, the only conceivable way in which a conflict can occur is from partitioning two slices, S_1 and S_2, that belong to faces L_1 and R_2 respectively, where L_1 and R_2 are left and right faces. Furthermore, L_1 and R_2 must be opposing faces because of the nesting property (proposition 3.1).

We note that the set of slices belonging to some face is consecutive. Say L_1 dominates R_2 (the other case is symmetrical). Then the right endpoint of a segment created from a slice belonging to L_1 is at distance at most $l(L_1)$ from L_1 and the left endpoint of a segment created from a slice belonging to R_2 is at distance at most $l(R_2)-1$ from R_2. Now we apply lemma 3.10: the distance between L_1 and R_2 is at least $l(L_1)+l(R_2)$. Therefore all right endpoints of segments created from slices belonging to L_1 are less than all left endpoints of segments created from slices belonging to R_2, so no conflict can occur. □

3.4.3 Implementation Details

We now describe in detail a parallel implementation of the tournament construction algorithm described above. Our algorithm works in time $O(\log n)$ and uses $O(n^2/\log n)$ processors on a concurrent read - exclusive write (CREW) PRAM, where n is the number of vertices in the tournament. Our parallel algorithm is optimal, since the size of the output is $\Theta(n^2)$. Some of the procedures will be easier to describe as using $O(n^2)$ processors and working in constant time. Each such procedure can clearly be slowed down to work in time $O(\log n)$ using only $O(n^2/\log n)$ processors.

Let U be the upset polygon corresponding to the input score list. The area of U (i.e. the number of squares it contains) can be $\Theta(n^3)$, since its height can be $\Theta(n^2)$ (for example, the area of an upset polygon of a a regular tournament is $\frac{(n-1)n(n+1)}{12}$). The first step we perform is to "compress" U to get an $O(n^2)$ representation.

Let $l_1<l_2<\cdots<l_m$ be the sorted list of values of the upset sequence $\vec{u}$ (l_i is the i'th smallest u value). The *i'th level* of U is the sub-polygon with y-coordinates between $l_{i-1}+1$ and l_i (where $l_0\equiv 0$). It is easy to see that each level

is a collection of *rectangles*. In other words, for every column j and level r, squares in j appear either in all the heights of r or in none of them. We can, thus, talk about "slices at level r". We represent U by a zero-one matrix, *LEVEL*, where $LEVEL[r,j]=1$ if and only if $u_j \geq l_r$. For a complete description we also keep a vector *HEIGHT*, where $HEIGHT[r]$ is the height of the highest slice in level r. *LEVEL* can be computed using $O(n^2)$ processors, each computing one entry in constant time.

We now list the steps of the computation. In each step a matrix or vector is computed, and in the final step a processor is assigned to each slice and 2-partitions it. We start by listing the matrices and vectors computed and then describe in detail how each step is implemented.

A vector *TOP_LEVEL*. $TOP\_LEVEL[k]$ is the maximum level, r, such that $LEVEL[r,k]=1$ (i.e. the highest level of column k).

A matrix *ENDPOINT*. If there is a slice $[i,j]$ in level r, then $ENDPOINT[r,j]=i$ and $ENDPOINT[r,i]=j$. If no slice begins or ends at column j in level r then $ENDPOINT[r,j]=empty$.

Matrices *TOP* and *BOTTOM*. $TOP[i,j]$ is the top level in which slice $[i,j]$ appears. $BOTTOM[i,j]$ is the bottom level in which slice $[i,j]$ appears. (Again, an entry is *empty* if no such slice exists).

Face domination matrix, *FD*. $FD[i,j]=1$ if face j dominates face i. $FD[i,j]=0$ if face i dominates face j. $FD[i,j]=empty$ if faces i and j are not opposing. (See section 3.4.2 for the definition of face domination.)

Vector *TOP_SLICE*. $TOP\_SLICE[k]$ is the level of the highest slice that belongs to face k (the face between columns $k-1$ and k).

TOP_LEVEL can be computed in constant time by assigning a processor to each entry of *LEVEL* to check if it is 1 and the entry above it is 0.

The r'th row of *ENDPOINT* is computed using $O(n/\log n)$ processors in $O(\log n)$ time by a balanced binary tree computation ([MR]). We "plant" a balanced complete binary tree with $n-1$ leaves on level r of the upset polygon. Each node, N, in the tree represents a range of entries in row r of *LEVEL*, between columns $l(N)$ and $r(N)$. A node computes three functions:

propagate(N) - is true iff all the entries represented by N are 1.

start_right(N) - the first column of a slice starting between $l(N)$ and $r(N)$ and ending to the right of $r(N)-1$.

end_left(N) - the last column of a slice ending between $l(N)$ and $r(N)$ and starting to the left of $l(N)+1$.

An internal node, N, has two children, N_{left} and N_{right}, where $l(N_{left})=l(N)$, $r(N_{right})=r(N)$ and $r(N_{left})=l(N_{right})-1$. Then we have:

propagate(N) = *propagate*(N_{left}) and *propagate*(N_{right}).

start_right(N) = if *propagate*(N_{right}) then *start_right*(N_{left}) else *start_right*(N_{right}).

end_left(N) = if *propagate*(N_{left}) then *end_left*(N_{right}) else *end_left*(N_{left}).

The leaves of the tree represent single entries. If an entry is 0 then *propagate* = false and *end_left* and *start_right* are both *empty*. If an entry is 1 then *propagate* = true and *end_left* and *start_right* are both j (for the leaf representing entry j). A node computes its functions after its children have computed theirs. Furthermore, N, writes *end_left*(N_{right}) in *ENDPOINT*[*start_right*(N_{left}) and *start_right*(N_{left}) in *ENDPOINT*[*end_left*(N_{right}). Note that a value may be overwritten several times. After completing computing the functions for the whole tree, for each entry, j, if $LEVEL[r\,j-1]=1$ and $LEVEL[r\,j+1]=1$, then $ENDPOINT[r\,j]$ is set to *empty*.

It takes $O(\log n)$ time for the node functions to be evaluated for the entire tree. The whole computation can be done with $O(n/\log n)$ processors by a standard load-balancing trick, as described in [MR]. Proof that this procedure works correctly is straightforward, and is omitted.

TOP and *BOTTOM* are computed by having a processor for each entry of *ENDPOINT*. Processor [r,i] writes "j" in $TOP[i\,j]$ if $ENDPOINT[r,i]=j$ and $ENDPOINT[r+1,i]\neq j$. Similarly for *BOTTOM*.

$FD[i\,j]=1$ if $ENDPOINT[TOP[i\,j]+1\,j]=empty$ and either $ENDPOINT[TOP[i\,j]+1\,j]\neq empty$ or $i<j$).

For computing *TOP_SLICE*, let $t=ENDPOINT[TOP\_LEVEL[k],k]$. Then $[k,t]$ is the highest slice touching face k. If $FD[k,t]=1$ then $TOP\_SLICE[k]$ is equal to $TOP\_LEVEL[k]$. Otherwise, it is one level below $BOTTOM[k,t]$ (unless

face k has no other slices than $[k,t]$, which can be checked by looking up $LEVEL[BOTTOM[k,t]-1,k])$.

Finally we partition each of the slices. Let $s=[l,r;h]$ be a slice. We use FD to find if s belongs to face l or face r. Then we use TOP_SLICE and $HEIGHT$ to find the height, h', of the highest slice belonging to that face. Now we can partition s according to its height, h, and h' as described in theorem 3.5.

We need to show how to assign processors to slices. One way to do it is as follows: a vector, V, is created with one entry for each left face, with the entry being the length of the face. A vector, P, of partial sums of V is computed. This vector contains, essentially, an enumeration of the slices. Let α be the total number of slices of U. We assign $\log n$ consecutive slices to each of $\alpha/\log n$ processors. Each processor finds its first slice in time $O(\log n)$ by a binary search on P. After that, each of the successive slices is accessed in constant time.

CHAPTER FOUR

STRONG ORIENTATION OF MIXED GRAPHS AND RELATED AUGMENTATION PROBLEMS

4.1 Introduction

The *strong orientation problem* is a problem in graph theory that stems from the following question - can the streets of a city be all changed into one-way streets so that every point is reachable from any other? There are two variants of this problem: the first variant is when the input graph, G, is undirected (i.e. the city initially contains only two-way streets). A more general situation is when G is mixed, i.e. contains some directed arcs and some undirected edges. In both cases, the problem is to assign orientations to the undirected edges of G to yield a strongly connected digraph. If G is mixed we require that the directions of the arcs of G are not altered by the orientation. If G admits such an orientation we say it is *strongly orientable.*

What if G is not strongly orientable? We define the *minimum strong augmentation problem:* find a minimum set of arcs (or edges) whose addition to G will make it strongly orientable. Notice that we need only consider addition of arcs, since in a strong orientation each edge is replaced by an arc. It is interesting to consider the special cases when G is either completely directed or completely undirected. In the first case the problem becomes - add a minimum set of arcs to a digraph to make it strongly connected. When G is undirected the problem is (by Robbins' theorem, as we shall see) - add a minimum set of edges to make G bridge-connected, where a graph is *bridge−connected* if it is connected and has no bridges. In this context a *bridge* is an edge whose removal disconnects the graph.

The algorithms presented in this chapter all run in time $O(\log n)$ on a CRCW PRAM, where n is the number of vertices in the input graph. There are two main classes - algorithms for undirected graphs and algorithms for directed graphs. It turns out that all our algorithms for undirected graphs use $O(n+m)$ processors, where m is the number of edges in the input graph. Our algorithms for digraphs

involve computing transitive closure, and therefore require $O(M(n))$ processors, where $M(n)$ is the sequential time for multiplying two $n \times n$ matrices. A realistic bound is $O(n^3/\log n)$. Asymptotically less processors are required using fast matrix multiplication methods (which have, so far, all been parallelizable). The current best algorithm known works in time $O(n^{2.376})$ ([CW]). This difference between the efficiencies of algorithms of the two classes are a common phenomena in the literature, and the algorithms in this chapter are another example.

A basic operation that we use many times is finding strongly connected components, and it is not hard to show that the problems we solve are at least as hard as finding strongly connected components. A big open problem in the field is finding an algorithm to compute strongly connected components (or even, say, test if a digraph is acyclic) more efficiently then by computing transitive closure. Note that the linear-time sequential algorithm relies on depth-first search, which is not known to be in NC, or even RNC, for directed graphs. Therefore the algorithms we present are "optimal with respect to the state of the art".

The results presented in this chapter are:

(1) An NC algorithm for strongly orienting a mixed graph. Parallel algorithms that appeared previously in the literature have been for strongly orienting *undirected* graphs.

(2) NC algorithms for constructing a minimum-cardinality set of edges to make a graph bridge-connected, and a minimum-cardinality set of arcs to make a digraph strongly connected. The solution of the latter problem involves introducing the notion of a *dense matching*, which can be computed efficiently in parallel.

(3) An NC algorithm for constructing a minimum-cardinality set of arcs to augment a mixed graph into a strongly orientable mixed graph. Before our work no report of even a polynomial-time solution to this problem appeared in the literature. Independently of our research an optimal sequential algorithm has been published by Gusfield ([Gusf]).

A few words about related hard problems. One natural question is to find a strong orientation for which the diameter (i.e. maximum distance between two vertices) of the resulting digraph is minimized. Chvatal and Thomassen ([CT]) have shown that even the problem of deciding if there exists an orientation for which

the directed diameter is equal to 2 is NP-hard. On the other hand they show that any graph with diameter d admits an orientation with diameter at most $2d^2+2d$, and for diameter, d, there are graphs of arbitrary connectivity for which any orientation has diameter at least $d^2/4+d$.

Another interesting problem is to find a minimum-weight strong augmentation. Again this is NP-hard even in a very limited case - when the weights (of potential arcs to be added) are all either 1 or 2. This follows trivially from reductions in [ET] for both the directed and undirected case. By a similar reduction one can show that finding a minimum cardinality strong augmentation from a given subset of the arcs is NP-hard.

Definitions and notation: we define a *mixed graph* $G=(V,E,A)$ as follows: V is the *vertex set*; E is the *edge set*, where an edge is an unordered pair of vertices; A is the *arc set*, where an arc is an ordered pair of vertices. We denote the vertex, edge and arc sets of G by $V(G)$, $E(G)$ and $A(G)$ respectively. Given an edge, $e=\{u,v\}$, $e\in E(G)$, *orienting e* means deleting e from $E(G)$ and adding a new arc, a, to $A(G)$, where either $a=(u,v)$ or $a=(v,u)$. *Undirecting an arc*, $a=(u,v)$, means replacing it by the edge $e=\{u,v\}$. The *underlying undirected graph* of a mixed graph, G, is the graph obtained by undirecting all its arcs. *Directing an edge*, $e=\{u,v\}$, means replacing it by the pair of arcs (u,v) and (v,u). The *underlying directed graph* of a mixed graph, G, is the graph obtained by directing all its edges.

A *path from u to v* in a mixed graph, G, is a sequence of distinct vertices, $u=v_1, v_2,\ldots,v_k=v$, such that, for all $1\le i<k$, either $\{v_i,v_{i+1}\}\in E(G)$ or $(v_i,v_{i+1})\in A(G)$. We say that *v is reachable from u* if there is a path from u to v. A mixed graph is *connected* if every vertex is reachable from every other vertex (or, equivalently, its underlying directed graph is strongly connected).

We will be constructing several graphs based on a given mixed graph, $G=(V,E,A)$, which we define here:

$U(G) = (V,E)$ – The undirected part of G.

$D(G) = (V,A)$ – The directed part of G.

$S(G)$ – The strong component graph of G. This is a mixed multigraph (i.e. one in which there can be several arcs and edges joining two vertices) whose

vertices are the strongly connected components of $D(G)$. The edges and arcs are those going between strongly connected components of $D(G)$.

4.2 Background

4.2.1 Theorems and Sequential Complexity of Strong Orientation

The solution to the decision problem of strong orientation for undirected graphs was given by Robbins in 1939:

Theorem 4.1 (Robbins' theorem [Robbin]): An undirected graph is strongly orientable if and only if it is bridge-connected.

The necessity of this condition is clear, and it is not hard to prove that it is also sufficient. One proof of this theorem ([Rober]) is obtained by showing that if G is bridge-connected then the following yields a strong orientation: let T be a depth-first search tree of G; Orient all edges of T from a vertex to its child in T, and all other edges from a vertex to its ancestor. A nice consequence of this proof is that it provides a linear-time sequential algorithm for constructing a strong orientation of an undirected graph.

The solution for mixed graphs was given by Boesch and Tindell 41 years later, yet is very similar to Robbins' theorem.

Theorem 4.2 ([BT]): A mixed graph, G, is strongly orientable if and only if it is connected and its underlying undirected graph is bridge-connected.

Again, the conditions are clearly necessary, and the proof that they are sufficient is not complicated. One way to view this theorem is as follows: consider the underlying undirected graph, H, of G. Assume that we were given this graph initially, and have oriented some of its edges to obtain G. Then we can complete this orientation into a strong orientation if H was strongly orientable in the first place and if, in transforming H into G, we have not ruined connectedness.

This theorem gives rise to the following obvious sequential algorithm: orient the edges one by one. At each step assign the edge an orientation that maintains

connectedness of the graph. If no such orientation exists, the graph is not strongly orientable. Otherwise, the resulting digraph will be strongly connected. In [BT] it is remarked that an algorithm based on depth-first search exists for mixed graphs too. Indeed, such an algorithm (which achieves linear running time) appears in [CGT].

4.2.2 Parallel Algorithms for Undirected Strong Orientation

As indicated above, both variants of the strong orientation problem are efficiently solvable sequentially. Several papers have appeared about parallel algorithms for strong orientation of undirected graphs. Atallah [Atal] presented the problem as one of interest for parallel computation, and gave an $O(\log^2 n)$ solution using $O(n^3)$ processors on a CREW PRAM ($(O(\log n))$ on a CRCW PRAM). Two later papers give algorithms based on Atallah's method, and are aimed at reducing the number of processors. Tsin's algorithm [Tsin] runs in time $O(\log^2 n)$ using $O(n^2 / \log^2 n)$ processors on a CREW PRAM. Note that this gives an optimal processor-time product in the case of dense graphs. Vishkin [Vish2] gives two implementations, one that has the same complexity as that of Tsin, and one that runs in time $O(\log n)$ using $n+m$ processors on a CRCW PRAM (where m is the number of edges). A simplified algorithm with the same complexity as Vishkin's algorithm appears in [TV].

We give a brief high-level description of Atallah's algorithm, which is implemented in [Tsin] and [Vish2] as well. The first idea is to use an arbitrary spanning tree, T, as opposed to the depth-first spanning tree used in the sequential algorithm. Each edge not in T induces a fundamental cycle (which is the unique cycle consisting of that edge together with a subset of the edges of T). The observation is that if we orient its edges of each fundamental cycle in a consistent way along the cycle then the resulting orientation is strong. A problem might arise in that a tree edge might be contained in several fundamental cycles, and might receive conflicting orientations. For this end the idea of assigning *priorities* to cycles is used. Each fundamental cycle is given a distinct priority, and the orientation assigned to an edge is according to the fundamental cycle of highest priority that contains it. It is not hard to see that all edges can be assigned orientations in parallel, and an NC algorithm follows.

When the input graph is mixed, it is clear that we need a different method. The basic difference is that in the mixed case some of the edge orientations have been determined, so the way in which we can orient the rest of the edges is constrained. Furthermore, since $U(G)$ is not necessarily connected, there is no obvious analog to Atallah's algorithm that works for mixed graphs.

4.3 Strong Orientation of Mixed Graphs

Our algorithm works in three stages, after checking that the input graph is strongly orientable. This checking involves testing if the underlying undirected graph of G is bridge-connected and the underlying directed graph of G is strongly connected (the conditions of theorem 4.2).

In the first stage we orient a subset of the undirected edges according to directions imposed by existing directed arcs. If, for some arc, we find a cycle in which it lies, and orient the edges contained in that cycle in a consistent fashion along the cycle, then its endpoints will lie in the same strongly connected component. We want to do this in parallel for many arcs. For this we use the idea of assigning priorities mentioned above ([Atal]). In our case priorities are given to arcs of G. In order to gain efficiency, we do not apply this operation to all the arcs of G, but only to a "spanning forest" of $D(G)$, which we prove to be sufficient for our purposes. After completing this stage, the resulting mixed graph, G', is such that all directed arcs lie within strongly connected components.

In the second stage we orient undirected edges going between strongly connected components of $D(G')$ to obtain the graph G''. In this stage the only use of the directed arcs is in determining the strongly connected components. Our main theorem is that the graph $D(G'')$ (the directed subgraph of G'') is strongly connected.

In the third stage we simply assign an arbitrary orientation to any edge in G'' that is still undirected.

The algorithm is listed formally, followed by its complexity analysis and proof of correctness.

procedure *STRONG_ORIENTATION*(G)

(0) Check that the two conditions of theorem 4.2 for strong orientability hold for G. If not, abort.

(1)

(1.1) Find a spanning forest of the underlying undirected graph of $D(G)$. Call this set of arcs F.

(1.2) In parallel assign distinct integer priorities to all arcs of F: priority $f(a)$ for arc a.

(1.3) For all arcs, $a \in F$, do in parallel: let $a=(u,v)$. Find a simple path, p_a, from v to u in G. Assign each undirected edge in p_a a temporary orientation with priority $f(a)$ according to the direction of p_a. ((Note: edges with temporary orientations are still considered undirected.))

(1.4) For all undirected edges, e, do in parallel: if e has at least one temporary orientation, orient it according to the temporary orientation of highest priority.

Call the resulting graph G'.

(2) Construct the undirected multigraph $H=U(S(G'))$, (whose vertices are strongly connected components of $D(G')$ and whose edges are those going between strongly connected components).
Orient the edges of H by some strong orientation algorithm for undirected graphs (e.g [Vish2]). Call the resulting mixed graph G''.

(3) In parallel assign arbitrary orientations to all undirected edges of G''.

end *STRONG_ORIENTATION*

Complexity analysis and implementation details: We claim that our algorithm can be implemented to run in time $O(\log n)$ using $O(M(n))$ processors on a CRCW PRAM. (Recall that $M(n)$ is the number of processors required to multiply two matrices, or to compute the transitive closure of a matrix.) We indicate how each of the steps can be implemented with these resources.

Step (0): Testing if the underlying graph is bridge-connected can be done by finding

biconnected components [TV]. Testing if the graph is connected can be done by transitive closure.

Step (1.1): Finding a spanning forest can be done in time $O(\log n)$ using $O(n+m)$ processors (where n is the number of vertices of G and m is the total number of edges and arcs) by a modification of [SV] indicated in [TV].

Step (1.2): The arc (i,j) can be given the priority $im+j$. It is easy to see that no two arcs will get the same priority.

Step (1.3): The set of paths, p_a, can be found in the following way - construct matrices $P^{(0)},P^{(1)},\ldots,P^{(s)}$ where $s=\lceil \log_2 n \rceil$ as follows:

$$P^{(0)}[i,j] = \begin{cases} j & \text{if } (i,j) \in A(G) \text{ or } \{i,j\} \in E(G) \\ \varnothing & \text{otherwise} \end{cases}$$

$$P^{(r)}[i,j] = \begin{cases} P^{(r-1)}[i,j] & \text{if } P^{(r-1)}[i,j] \neq \varnothing \\ k & \text{if } P^{(r-1)}[i,j] = \varnothing,\ P^{(r-1)}[i,k] \neq \varnothing \text{ and } P^{(r-1)}[k,j] \neq \varnothing \text{ for some } k \\ \varnothing & \text{otherwise} \end{cases}$$

The meaning of these matrices is - if $P^{(r)}[i,j] = k$ then k is a vertex lying on a path from vertex i to vertex j such that the distances from i to k and from k to j are both no more than 2^r. Note that $P^{(r)}$ can be computed from $P^{(r-1)}$ using $O(M(n))$ processors in constant time on a common write PRAM. Now, a path from i to j can be reconstructed as follows - create an array $A[0{:}d]$, where $d=2^{s+1}$ (the smallest power of two strictly larger than n). Initially set $A[0]=i$, $A[d]=j$ and all other entries to zero. The array will be filled in $s+1$ steps (steps 0,1,...,s): in step r, if $A[x]\neq 0, A[y]\neq 0$ and for all z, $x<z<y$, $A[z]=0$, then set $A[(x+y)/2]=P^{(s-r)}[A[x],A[y]]$. In each step, all entries for that step can be filled in parallel in constant time, so the array can be filled in time $O(\log n)$ with $O(n)$ processors. Reading the array from index 0 to index d gives the path from i to j with possible repetitions of vertices (i.e. each vertex of the path appears in a consecutive set of indices of the array).

Step (1.4): This can be done in constant time on a PRIORITY CRCW PRAM using $O(n^2)$ processors - have a processor for each index of each of the arrays generated

in step (1.3). Say processor p is in charge of index l of the array A, corresponding to the path from i to j. p checks if $A[l] \neq A[l+1]$. If so, and if $\{A[l],A[l+1]\}$ is an undirected edge of the graph, then p writes the direction $(A[l],A[l+1])$ into the memory cell designated for holding the orientation of the edge $\{A[l],A[l+1]\}$. This can be simulated in time $O(\log n)$ with the same number of processors by any other PRAM model using standard simulations.

Steps (2) and (3): After constructing $U(S(G'))$ (transitive closure), Vishkin's algorithm ([Vish2]) can be used to compute the orientation. Step (3) is clearly easy.

Next we prove the correctness of the algorithm. For simplicity of presentation we assume that the the input graph, G, is strongly orientable (i.e. that it passes the tests of steps (0)).

Lemma 4.1: Let $e = \{u,v\}$ be an undirected edge, which becomes oriented at stage 1 of the algorithm. Then u and v belong to the same strongly connected component of $D(G')$.

Proof: It is helpful to view stage 1 as happening in phases: first orient all edges with temporary orientation of the highest priority; next orient all unoriented edged with second highest priority, and so on. We proceed to prove the claim stated in the lemma by induction on $f(a)$, the temporary orientation with highest priority that e receives in step 1.2.

The base case is when $f(a)$ is the highest priority over all arcs of G. In this case all edges of p_a are oriented in a consistent fashion along some cycle, so following this u and v lie on a directed cycle, thus in the same strongly connected component.

For the induction step assume that the claim holds for all edges with temporary orientation of higher priority than $f(a)$. If all edges in p_a are oriented to form a cycle with a then, again, u and v will lie on a cycle. Assume some edges, $g_i = \{w_i,x_i\}$, in p_a have been oriented in previous phases counter to the direction of p_a. By the induction hypothesis, for all i, w_i and x_i are mutually reachable via directed paths. Thus after the current phase there will be directed paths from v to u and from u to v. □

Lemma 4.2: Let $a = (u,v)$ be an arc in the spanning forest, F, computed in step (1.1). Then u and v belong to the same strongly connected component of $D(G')$.

Proof: Similar to the proof of lemma 4.1. □

Lemma 4.3: The digraph $D(G')$ is either strongly connected, or consists of several isolated strongly connected components.

Proof: Let $a=(u,v)$ be an arc of G'. If a was an arc of G then, u and v were connected by some (not necessarily directed) path in the spanning forest, F, found in step (1.1). The fact that a lies inside a strongly connected component of $D(G')$ follows from repeated application of lemma 4.2. If a was not an arc of G, it must have been an edge that was oriented in stage 1, and by lemma 4.1 lies inside a strongly connected component of $D(G')$. □

Lemma 4.4: The mixed graph, G', is strongly orientable.

Proof: The second condition of theorem 4.2 clearly holds: the underlying undirected graph of G' is the same as that of G, and thus has no bridges. Assume the first condition is violated, i.e. there exist a pair of vertices, u and v, such that there is a (mixed) path, p, from u to v in G, but not in G'. Let x be the last vertex in p that is reachable from u in G', and let y be the first nonreachable vertex. It follows that $\{x,y\}$ is an undirected edge in G that was oriented in stage 1 to become (y,x) in G'. But by lemma 4.1 there is a (directed) path from x to y in G'. A contradiction. □

Lemma 4.5: The undirected multigraph, $U(S(G'))$ (whose vertices are the strongly connected components of $D(G')$ and edges are undirected edges between components) is strongly orientable.

Proof: By lemma 4.3, the only edges between strongly connected components of G' are undirected. Thus $U(S(G'))=S(G')$. By lemma 4.4, G' is strongly orientable, so, clearly, $S(G')$ must also be strongly orientable. □

Theorem 4.3: The directed graph, $D(G'')$ is strongly connected.

Proof: By lemma 4.5, stage 2 of the algorithm yields a strong orientation of $S(G')$. Thus in the resulting directed graph, $D(G'')$, every vertex is reachable from any other. □

4.4 Minimum Strong Augmentation of Graphs and Digraphs

The problems we consider in this section are: find a minimum augmentation to make an undirected graph bridge-connected and to make a digraph strongly connected. Linear time sequential algorithms for both these problems appear in [ET]. Our parallel solution for the undirected case is a fairly straightforward parallelization of that in [ET]. Our solution in the directed case is quite different.

4.4.1 Making a Graph Bridge-Connected

The algorithm for making an undirected graph, G, bridge-connected is essentially that of [ET]. The purpose of this section is to show that this algorithm has a fast and efficient parallel implementation and to obtain some insight towards solving the problem for mixed graphs. The proof of correctness of the algorithm below appears in [ET], and will not be repeated here.

procedure *UNDIRECTED_AUGMENTATION*(G)

(0) Set $A = \emptyset$. ((A is the augmenting set of edges.))

(1) Construct, $BC(G)$, the bridge-component graph of G: vertices are bridge-connected components of G and edges are edges between bridge-connected components. ((Note that $BC(G)$ is a forest.))

(2) Let $T_1,...,T_k$ be the connected components of $BC(G)$. Let L_{2i-1} and L_{2i} be distinct leaves of T_i for all i (or the same leaf if T_i is an isolated vertex). In parallel, for all $i<k$, add to A an edge from some vertex of the bridge-connected component L_{2i} to some vertex of L_{2i+1}.

(3) Let G' be the augmented graph after step (2). Note that $BC(G')$ is a tree. Pick some non-leaf vertex, r, of $BC(G')$ and root $BC(G')$ at r.

(4) Number the vertices of $BC(G')$ in preorder.

(5) Let $L(1),L(2),...,L(m)$ be the leaves of $BC(G')$ in increasing preorder number. For all $i \leq \lfloor m/2 \rfloor$ do in parallel: add to A an edge from some vertex of the bridge-connected component $L(i)$ to some vertex of $L(i+\lfloor m/2 \rfloor)$.

(6) Return A.

end *UNDIRECTED_AUGMENTATION*

Complexity analysis and implementation details: We indicate how the procedure can be implemented to run in time $O(\log n)$ using $O(n+m)$ processors on a CRCW PRAM, where the input graph, G, has n vertices and m edges: finding the bridge-connected components of G can be done by finding biconnected components ([TV]). This is because the bridges of G are exactly the blocks containing a single edge. The bridge-components are the connected components of G without its bridges. Rooting a tree and finding a preorder numbering can be done using the "Euler tour" technique of [TV] within the stated resource bounds.

4.4.2 Making a Digraph Strongly Connected

Let G be a digraph. A *source* of G is a vertex with no incoming arcs. A *sink* of G is a vertex with no outgoing arcs. Note that according to our definitions an isolated vertex is both a source and a sink. Recall that $S(G)$ is the acyclic digraph of the strongly connected components of G. An augmentation of $S(G)$ corresponds to an augmentation of G as follows: an arc from C_1 to C_2 in $S(G)$ corresponds to an arc from some vertex of the component C_1 to some vertex of the component C_2 in G. The following easily proven lemmas appear in [ET].

Lemma 4.6: An augmentation makes $S(G)$ strongly connected if and only if the corresponding augmentation makes G strongly connected.

Lemma 4.7: A lower bound on the number of arcs needed to make G strongly connected is:

$$sa(G) \equiv \begin{cases} 0 & \text{if } G \textit{ is strongly connected} \\ \max\{\textit{number of sources of } S(G), \textit{number of sinks of } S(G)\} & \textit{otherwise} \end{cases}$$

Our algorithm finds an augmenting set of arcs of size $sa(G)$, which strongly connects $S(G)$. By lemmas 4.6 and 4.7 this augmentation corresponds to a minimum augmentation of G. The following lemma allows us to focus our attention only on sources and sinks of $S(G)$:

Lemma 4.8: Let A be an augmentation that strongly connects all the sources and sinks of $S(G)$ (i.e merges them into one strongly connected component). Then A strongly connects $S(G)$.

Proof: Since $S(G)$ is acyclic, every vertex, v, of $S(G)$ lies on a path from some source, x, to some sink, y. Thus, in the augmented digraph, v lies on a closed trail containing x and y, and therefore is contained in the same strongly connected component as the sources and sinks of $S(G)$. []

Motivated by this lemma we define the undirected bipartite graph $B(G)=(X,Y,E)$ where:

X = set of sources of $S(G)$.

Y = set of sinks of $S(G)$.

$E = \{\{x,y\} \mid$ there is a (possibly empty) path from x to y in $S(G)\}$

Lemma 4.9: $B(G)$ has no isolated vertices.

Proof: An isolated vertex of $S(G)$ appears in $B(G)$ as a pair of vertices, $x \in X$ $y \in Y$, connected by an edge. For a source which is not isolated in $S(G)$, there must be a path from it to some sink, since $S(G)$ is finite. Similarly, every non-isolated sink is reachable from some source. []

It turns out that the notion of matching in the graph $B(G)$ is helpful in obtaining a minimum augmentation for strongly connecting G. A *maximal matching* is a matching that is not properly contained in any other matching.

Theorem 4.4: Let $M = \{\{x_1,y_1\}, \ldots, \{x_k,y_k\}\}$ be a maximal matching of $B(G)$, where x_i's are sources of $S(G)$ and y_i's are sinks. Let $A = \{(y_1,x_2),\ldots,(y_{k-1},x_k),(y_k,x_1)\}$ be an augmentation of $S(G)$, and let G' be the augmented digraph. Then:

(1) $sa(G') = sa(G)-k$.

(2) In $S(G')$ every sink is reachable from every source.

Proof: The matching, M, together with the augmentation, A, constitute a spanning cycle of the vertices $x_1,y_1,\ldots,x_k,y_k$. Thus all these vertices are contained in

one strongly connected component of G'. Call this component C. If $S(G)$ has exactly k sources and k sinks (i.e M is a perfect matching) then G' is strongly connected (by lemma 4.8), and the theorem holds. Otherwise, if x is a source not covered by M then, by lemma 4.9, x has some neighbor, y, in $B(G)$. Since M is maximal, y must be covered by M (otherwise $M \bigcup \{x,y\}$ is a matching properly containing M). Therefore there is a path from x to C in G'. Similarly, if there is a sink not covered by M, there is a path from C to it in G'. Both parts of the theorem follow. □

The following lemma shows the importance of the second part of the theorem:

Lemma 4.10: Let G be an acyclic digraph in which every sink is reachable from every source, and let A be an augmentation of G in which every arc is from a sink to a source. If A contains at least one arc into every source and at least one arc out of every sink then A makes G strongly connected.

Proof: Picture the arcs of A as being added to G one by one, in an arbitrary order. The conditions of the lemma imply that after an arc is added, its endpoints lie in the same strongly connected component. Thus, after adding all the arcs of A, all the sources and sinks of G lie in the same strongly connected component, and by lemma 4.8, G becomes strongly connected. □

At this point we have enough ingredients for an NC algorithm for our problem: given G, construct $B(G)$ and find a maximal matching in it; augment G with an appropriate set of arcs according to theorem 4.4; finally, augment the resulting graph in the way indicated by lemma 4.10. Theorem 4.4 and lemma 4.10 imply that the augmentation is, indeed, of minimum size.

A careful look at the proof of theorem 4.4 reveals that the matching in the statement of the theorem need not be maximal. The property we really want is that every unmatched vertex has a matched neighbor. Let a *dense matching* be a matching for which every non-isolated vertex is is adjacent to a matched vertex. Then we have:

Corollary 4.1: Let M be a dense matching of $B(G)$. Then the statement of theorem 4.4 holds for M.

The distinction between a maximal matching and a dense matching is important for us, since we can find a dense matching in $O(\log n)$ parallel time, whereas the fastest processor-efficient algorithm known for finding a maximal matching runs in time $O(\log^3 n)$ ([IS]). The basis for a fast and efficient algorithm is the following lemma:

Lemma 4.6: Let F be a spanning forest of a graph, G. Then a dense matching of F is also a dense matching of G.

Proof: Let v be a non-isolated vertex of G. Then v is non-isolated in F. []

Our parallel algorithm follows.

procedure *DIRECTED_AUGMENTATION*(G)

(0) Set $A = \emptyset$. ((A is the augmenting set of arcs.))

(1) Find the strongly connected components of G and construct $B(G)$.

(2) Find M = *DENSE_MATCHING*$(B(G))$.

(4) Let $\{\{x_0,y_0\}, \ldots, \{x_{k-1},y_{k-1}\}\}$ be the edges of M. Put in A an arc from a vertex of the strongly connected component corresponding to y_i to a vertex of $x_{(i+1) \bmod k}$ for all i, $0 \leq i \leq k$.

(5) Let $a_0,\ldots,a_{h-1}$ be the sources uncovered by M and $b_0,\ldots,b_{l-1}$ be the sinks uncovered by M. Let $m = \min(h,l)$. Add to A an arc from a vertex of the strongly connected component corresponding to $b_{i \bmod m}$ to a vertex of $a_{i \bmod m}$ for all i.

(6) Return A.

end *DIRECTED_AUGMENTATION*

procedure *DENSE_MATCHING*(G)

((Returns a dense matching of an undirected graph, G))

(0) Set $M = \emptyset$. ((M is the matching.))

(1) Find a spanning forest, F, of G. The rest of the procedure is performed on all connected components of F in parallel.

(2) Let T be a connected component of F. Pick an arbitrary vertex, r, of T,

and make T rooted at r.

(3) For all vertices, v, of T in parallel, compute the distance of v from the root r.

(4) For all non-leaf vertices, v, of even distance from r do in parallel: pick one of v's children, u, and add $\{u,v\}$ to M.

(5) For all vertices, v, of odd distance from r do in parallel: if v is unmatched and it has an unmatched child then pick such a child, u, and add $\{u,v\}$ to M.

(6) Return M.

end *DENSE_MATCHING*

Lemma 4.12: The set of edges, M, computed by procedure *DENSE_MATCHING* is a dense matching in the given graph, G.

Proof: That M is a matching follows from the fact that the parent in a tree is unique. By lemma 4.11, we need only show that M is dense in every component of the spanning forest, F, of G. If v is a vertex of odd distance from the root of its component then its parent is matched at step (4). If v is a non-leaf vertex of even distance from the root then it is matched to one of its children in step (4). If it is a leaf of even distance then its parent is matched either in step (4) or in step (5). []

Complexity analysis and implementation details: First we show that *DENSE_MATCHING*(G) runs in time $O(\log n)$ using $O(n+m)$ processors on a CRCW PRAM, where n is the number of vertices of G and m is the number of edges: A spanning forest, F, can be computed with these bounds by a modification of the Shiloach-Vishkin algorithm ([SV]) indicated in [TV]. Rooting a tree at a vertex can be done using the "Euler tour" technique of [TV]. Finding distances from the root is performed by a standard "doubling" technique - $\lceil \log n \rceil$ stages, where at stage k every vertex points to its ancestor of distance 2^k (or to the root, if it is closer than 2^k). Steps (4) and (5) are very local in nature, and can clearly be executed with the stated resources (by having vertices try to match-up with their parents, and solving conflicts by the concurrent write feature).

Finally we observe that the only expensive step in *DIRECTED_AUGMENTATION* is step (1), which can be done by a transitive clo-

sure computation, in time $O(\log n)$ using $O(M(n))$ processors.

4.5 Minimum Strong Augmentation of Mixed Graphs

In this section we give an NC algorithm for the problem of finding a minimum augmenting set of arcs for a given mixed graph, G, such that the resulting mixed graph is strongly orientable. By theorem 4.2 we can phrase the problem as:

First formulation: Augment G with a minimum set of arcs such that:

(1) The underlying directed graph of the augmented graph is strongly connected.

(2) The underlying undirected graph of the augmented graph is bridge-connected.

From the previous section we know that in order for an augmentation to accomplish (1) it needs to contain arcs incident to all sources and sinks of the strong component graph of the underlying directed graph of G (to be called "super sources" and "super sinks" below). In order for an augmentation to accomplish (2) it needs to contain arcs incident to all leaves of the bridge-component graph of the underlying undirected graph of G (to be called "super leaves"). A hard aspect of the problem in this formulation is that super sources and sinks can intersect super leaves in many ways. One can find a minimum set of vertices hitting all the super sources, sinks and leaves, but it is not clear how to proceed after finding such a set.

A different formulation leads to our solution. Recall that, for a digraph, D, $sa(D)$ is the maximum of the number of sources of $S(D)$ and the number of sinks of $S(D)$ (or zero, if D is strongly connected).

Second formulation: Orient the edges of a given mixed graph, G, such that for the resulting digraph, D, $sa(D)$ is minimized.

Lemma 4.13: The second formulation yields a minimum strong augmentation for G.

Proof: We can view our task as augmenting G to become strongly orientable and then orienting its edges to obtain a strongly connected digraph. If we switch the

order of the operations - first orient to obtain D and then augment D, we know that the size of the required augmentation is exactly $sa(D)$. Thus a minimum augmentation is one for which $sa(D)$ is minimized. []

The first step of our algorithm is to orient the "strongly orientable" edges of G. We say that $e=\{u,v\}$ is *strongly orientable* if there is a path from u to v or from v to u in $G-\{e\}$. We can orient these edges by applying to G the strong orientation algorithm of section 3 with the simple modification of skipping steps (0) and (3). After this step the endpoints of each strongly orientable edge lie on a directed cycle, so the partial orientation obtained is clearly optimal. The proof of correctness of this step is similar to the proofs of section 2.

Let G' be the mixed graph obtained after the first step. Let $F=U(S(G))$ (recall that this is the undirected part of the mixed graph whose vertices are the strongly connected components of $D(G')$, the directed part of G'). Since there are no strongly orientable edges in G' it follows that F is a forest. Furthermore, orienting edges of F will not create new directed cycles, and thus we have:

Lemma 4.14: Let G'' be any digraph obtained by orienting the edges of G'. Then the strongly connected components of G'' are the same as those of $D(G')$.

Our goal is to find an orientation that minimizes $sa(G'')$. Lemma 4.14 says that by orienting edges of F we will not change the strong component structure of the graph. What we *can* do is to possibly decrease the number of sources and sinks. For example, if F contains an edge between a source and a sink we can orient it from the sink to the source, thus eliminating one source and one sink.

We state our problem in terms of supply and demand. Each vertex of F has one of four possible labels:

I - a source vertex (demanding an arc Into it)

O - a sink vertex (demanding an arc Out of it)

IO - an isolated vertex, which is both a source and a sink

X - a vertex with no demands (neither a source nor a sink).

A vertex is *unsatisfied* if the orientation does not provide it with the arc(s) it demands. Our task is to orient the edges of F so as to minimize

max {# of unsatisfied I's, # of unsatisfied O's}.

Our approach is the following - let T be a connected component of F. We compute the following numbers of sources and sinks among vertices of T remaining after orienting the edges of T:

$i(T)$ = the minimum possible number of unsatisfied sources

$o(T)$ = the minimum possible number of unsatisfied sinks

$t(T)$ = the minimum possible total number of unsatisfied sources and sinks.

These numbers can be computed by a simple case analysis of the labels of vertices of T. The case analysis appears in the listing of the algorithm below. After calculating $i(T)$, $o(T)$ and $t(T)$ for all components of F, we perform a simple global analysis to decide, for each component, which of its vertices are to remain sources and which are to remain sinks and orient its edges accordingly. Finally we augment the resulting digraph using the procedure *DIRECTED_AUGMENTATION* of the previous section.

procedure *MIXED_AUGMENTATION*(G)

(1) Apply to G steps (1) and (2) of the strong orientation algorithm of section 3. Call the resulting mixed graph G'.

(2) Compute $F=U(S(G'))$ and label each vertex of F by one of four labels - I,O,IO,X for source, sink, isolated vertex and other, respectively.

(3) For all connected components, T, of F do in parallel:

(3.1) Let k be the number of leaves of T labeled IO.

(3.2) If all vertices of T are IO then mark $k-2$ IO leaves as "free" and set

$$i(T)=1,\ o(T)=1,\ t(T)=k$$

(3.3) If all vertices of T are I or IO (at least one I) then mark $k-1$ IO leaves as "free" and set

$$i(T)=1,\ o(T)=0,\ t(T)=\max\{k,1\}$$

(3.4) If all vertices of T are O or IO (at least one O) then mark $k-1$ IO leaves as "free" and set

$$i(T)=0,\ o(T)=1,\ t(T)=\max\{k,1\}$$

(3.5) In all other cases mark all k IO leaves as "free" and set

$$i(T)=0,\ o(T)=0,\ t(T)=k$$

(4) Let

i = sum of $i(T)$'s of all connected components, T, of F.

o = sum of $o(T)$'s of all connected components, T, of F.

t = sum of $t(T)$'s of all connected components, T, of F.

If $i \geq o$ then set $N_i = \max\{\lceil t/2 \rceil, i\}$, and $N_o = t - N_i$.

If $i < o$ then set $N_o = \max\{\lceil t/2 \rceil, o\}$, and $N_i = t - N_o$.

((Remark: N_i and N_o are, respectively, the number of sources and sinks in the resulting digraph.))

(5) Mark the first $N_i - i$ "free" vertices of F as "designated source", and all the other "free" vertices as "designated sink".

(6) For all connected components, T, of F do in parallel: *ORIENT_LABELED_TREE*(T).

(7) Let G'' be the resulting digraph from the steps so far. Compute A = *DIRECTED_AUGMENTATION*(G'').

(8) Return A.

end *MIXED_AUGMENTATION*

procedure *ORIENT_LABELED_TREE*(T)

((Orients a tree with labels on its vertices.))

(1) Find a root, R, according to the following cases:

(1.1) If T has an X vertex, v, set $R = v$.

(1.2) Else, if T has an I or "designated sink" vertex, v, and an O or "designated source" vertex, u, then orient the path, p, between v and u in the direction from u to v, and set $R = p$.

(1.3) Else, select an arbitrary non-leaf, v, and set $R = v$.

(2) Root T at R.

(3) In parallel do for each edge, e, which has not been oriented in step (1): if any leaf in the subtree below e is labeled either I or "designated sink", then orient e away from the root, R. Otherwise orient e towards the root.

end *ORIENT_LABELED_TREE*

Lemma 4.15: After orienting a tree, T, by procedure *ORIENT_LABELED_TREE*, the total number of remaining sources and sinks is $t(T)$, and vertices marked "designated sources" and "designated sinks" become, respectively, sources and sinks.

Proof: A simple inductive proof shows that the only vertices which remain as sources or sinks after the orientation are *IO* leaves and, if selected at step (1.3), the root, R. One can verify that the number of such vertices is $t(T)$. Furthermore, it is not hard to see that except for $i(T)$ vertices which remain as sources and $o(T)$ vertices which remain as sinks, all the vertices are either satisfied or become what they are designated to be. □

Theorem 4.5: The procedure *MIXED_AUGMENTATION* computes a minimum augmenting set of arcs that makes a given mixed graph, G, strongly orientable.

Proof: The discussion before the algorithm listing proves that the general scheme is correct. We leave out the simple yet tedious proof that the i, o and t values computed in step 3 are accurate. Given these numbers, it is clear that the minimum total number of sources is i, the minimum total number of sinks is o and the minimum total number of sources and sinks is t. Thus the optimal way to designate sources and sinks is to try to have no more than $\lceil t/2 \rceil$ sources and no more than $\lceil t/2 \rceil$ sinks. This is what we do in steps (4) and (5) within the bounds set by i and o. Finally, lemma 5.3 shows that the numbers of sources and sinks remaining after orienting the edges of F, are, indeed, the desired numbers. []

Complexity analysis and implementation details: First we show that *ORIENT_LABELED_TREE* runs in $O(\log n)$ time using $O(n+m)$ processors: steps (1) and (2) can be done using the "Euler tour" technique of [TV]. Step (3) can be implemented by the tree contraction method of Miller and Reif ([MR]). Next we note that all steps except (1) of *MIXED_AUGMENTATION* can also be implemented with $O(n+m)$ processors, since the most complicated operation is computing connected components. Finally, the most expensive step is step (1), which was shown (in section 4.3) to require $O(\log n)$ time and $O(M(n))$ processors.

CHAPTER FIVE

ZERO-ONE SUPPLY-DEMAND PROBLEMS

5.1 Introduction

Supply-demand problems are fundamental in combinatorial optimization ([FF],[Lawler]). In one formulation of the problem the input is a network in which each arc has a non-negative capacity, and each vertex has a certain supply or demand (possibly zero). The task is to find a flow function, such that the flow through each arc is no more than its capacity and the difference between the flow into a vertex and out of it is equal to its supply (or demand). This problem is equivalent to the general max flow problem, and can, therefore, be solved efficiently sequentially ([Lawler],[PS],[GT]), but probably has no efficient parallel solution, since it is P-complete ([GSS]). There are, however, many interesting special cases of this problem whose solutions do not require the full power of general max flow.

In this chapter we are concerned with several such problems. The first problem we discuss is: given a sequence of supplies, $a_1, \ldots, a_n$, and demands, $b_1, \ldots, b_m$, construct a zero-one flow pattern satisfying these constraints, where every supply vertex can send at most one unit of flow to each demand vertex. Equivalently, we can state this problem as that of constructing a zero-one matrix, M, having a_i 1's in the ith row and b_j 1's in the jth column (for all $1 \le i \le n$, $1 \le j \le m$). We will refer to this problem as the *matrix construction problem*. M is called a *realization* for the input $(\vec{a},\vec{b})$. There is a simple sequential algorithm for constructing a realization if one exists ([FF],[Gale]): select any row, assign its 1's to the columns having largest column sums and repeat this procedure in the reduced problem. If this procedure gets stuck (i.e. some column sum becomes negative), then no realization exists. Note that this procedure is similar to the one described in section 3.1 for constructing a tournament with a specified degree sequence.

This algorithm, although easy to implement sequentially, seems very hard to

parallelize. Thus it is natural to ask if there is a fast parallel algorithm for this problem. Two remarks are relevant to this question: first, the problem can be solved by network flow techniques. Since the capacities are small (polynomial in the size of the flow network), there are *Random NC* algorithms for the problem by reduction to maximum matching ([KUW2],[MVV]). Second, there is a simple sequential method for *testing* whether an instance, $(\vec{a},\vec{b})$ is realizable ([FF],[Berge]). It is based on partial sums of the sequences, and can be implemented in *NC* in a straightforward manner. However, this method does not yield a way of *constructing* a realization. This is another example of the apparent gap between search and decision problems in the parallel realm ([KUW1]).

We present a deterministic *NC* algorithm for the matrix construction problem. Our algorithm can be implemented to run in time $O(\log^4|M|)$ using $O(|M|\cdot(n+m))$ processors on a CRCW PRAM, or in time $O(\log^3|M|)$ using $O(|M|\cdot(n+m)^3)$ processors on an EREW PRAM, where M is the realization matrix with n rows and m columns and $|M|$ is the size of M (i.e. $n\cdot m$). When $n=\Theta(m)$ the number of processors is $O(|M|^{1.5})$. and $O(|M|^{2.5})$ respectively.

The algorithm is based on a careful examination of the network flow formulation of the problem. It exploits the fact that there are only a polynomial number of cuts which need to be considered, and that this set of potentially min cuts has a natural ordering associated with it.

The methodology we develop enables us to solve the following two related problems (with the same time and processor bounds):

(1) The *symmetric supply−demand problem* - given a sequence of positive and negative integers summing to zero, representing supplies and demands respectively, construct a zero-one flow pattern so that the net flow out of (into) each vertex is its supply (demand), where every vertex can send at most one unit of flow to every other vertex. Notice that this problem is quite different than the matrix construction problem, since it does not have a "bipartite" nature.

(2) The *digraph construction problem* - construct a *simple* directed graph with specified in- and out-degrees. This corresponds to constructing a zero-one matrix with specified row and column sums, where the diagonal entries are forced to be zero. [FF] and [Berge] give a simple sequential algorithm when the in- and out-

degrees are sorted in the same order (i.e. a vertex with higher in-degree has higher out-degree). Our algorithm is the only one we know of for general orders That does not use max flow.

In the last section we extend our results to the case where the input represents upper bounds on supplies and lower bounds on demands.

5.2 The Matrix Construction Problem

5.2.1 The Slack Matrix

Our parallel algorithm is based on a careful analysis of the network flow formulation of the problem. The main tool we use is, what we call, the **slack matrix** which is similar to the "structure matrix" of Ryser [Ryser]. In order to define the slack matrix, we need to look at the solution to our problem by network flow. Given the input $(\vec{a},\vec{b})$: $a_1 \geq a_2 \geq \cdots \geq a_n$, $b_1 \geq b_2 \geq \cdots \geq b_m$, we construct a flow network, N, as shown in fig. 5.1: the vertex set consists of a source, s, a sink, t, vertices u_i , $1 \leq i \leq n$ corresponding to rows and vertices v_j , $1 \leq j \leq m$ corresponding to columns. The arc set contains three types of arcs: for all $1 \leq i \leq n$, $1 \leq j \leq m$ there are arcs (s,u_i) of capacity a_i, (v_j,t) of capacity b_j and (u_i,v_j) of capacity 1.

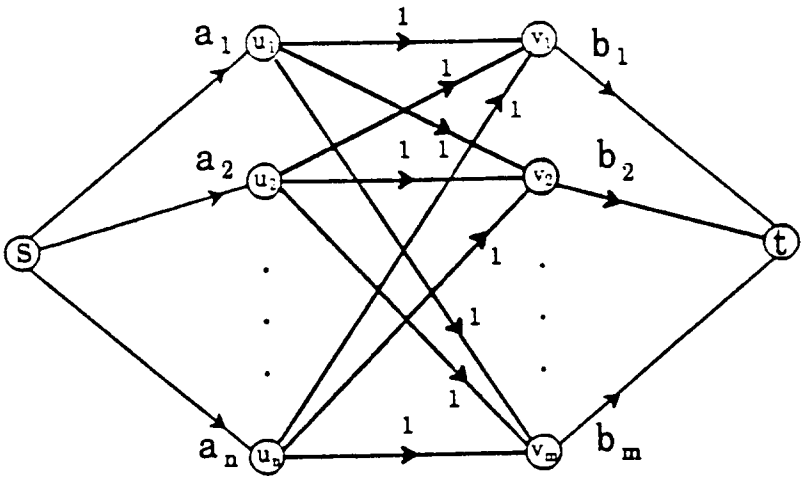

Fig. 5.1 : Flow network for solving the 0-1 matrix construction problem

Let $S = \sum_{i=1}^{n} a_i = \sum_{j=1}^{m} b_j$. Clearly the max flow value in N is bounded by S. Furthermore, a flow which satisfies all rows and columns sums is of value S. It follows

(by the max flow - min cut theorem) that the problem instance $(\vec{a},\vec{b})$ is realizable if and only if every directed cut in N has capacity at least S.

Let $C = (C^s:C^t)$ be a directed cut in N (i.e. the vertices are partitioned into two sets, C^s, C^t such that $s \in C^s$, $t \in C^t$). Say C^s contains x vertices from the set $\{u_1, \ldots, u_n\}$ and $m-y$ vertices from $\{v_1, \ldots, v_m\}$. Observe that if we replace u_j by u_i in C^s, for some $i<j$, then the capacity of the cut can only decrease. Similarly, replacing v_k by v_l in C^s can only decrease the capacity of the cut, for $l>k$. It follows that the capacity of C is no less than the capacity of the cut $C_{x,y}$, where $C^s_{x,y} = \{s\} \bigcup \{u_1,\ldots,u_x\} \bigcup \{v_{y+1},\ldots,v_m\}$. Thus there are only $n \cdot m$ cuts, $\{C_{x,y} \mid 1 \le x \le n \ , \ 1 \le y \le m\}$, which are potential min cuts. The cut $C_{x,y}$ is shown in fig. 5.2. Therefore, necessary and sufficient conditions for the instance $(\vec{a},\vec{b})$ to be realizable are that for every $1 \le x \le n \ 1 \le y \le m$:

$$capacity(C_{x,y}) = \sum_{i=x+1}^{n} a_i + \sum_{j=y+1}^{m} b_j + x \cdot y \ge S$$

$$\sum_{i=x+1}^{n} a_i + (S - \sum_{j=1}^{y} b_j) + x \cdot y \ge S$$

$$\sum_{i=x+1}^{n} a_i - \sum_{j=1}^{y} b_j + x \cdot y \ge 0$$

Definition: The *slack* of $C_{x,y}$ of problem instance $(\vec{a},\vec{b})$ is:

$$sl_{\vec{a},\vec{b}}(x,y) = \sum_{i=x+1}^{n} a_i - \sum_{j=1}^{y} b_j + x \cdot y$$

The *slack matrix*, $SL_{\vec{a},\vec{b}}$, is the matrix whose i,jth entry is $sl_{\vec{a},\vec{b}}(i,j)$.

Proposition 5.1: The instance $(\vec{a},\vec{b})$ is realizable if and only if $SL_{\vec{a},\vec{b}}$ is non-negative.

Proposition 5.2: Let $(\vec{a},\vec{b})$ be an instance which is realizable by some matrix, M, and assume that $sl_{\vec{a},\vec{b}}(x,y) = 0$. Then:

$$(1)\ M[i,j]=1 \quad \text{for } all \quad 1 \le i \le x \ , \ 1 \le j \le y$$

$$(2)\ M[i,j]=0 \quad \text{for } all \quad x+1 \le i \le n \ , \ y+1 \le j \le m$$

Proof: Since $sl_{\vec{a},\vec{b}}(x,y) = 0$, the cut $C_{x,y}$ has capacity S, which means that in any max flow forward arcs (1) are all saturated, and backward arcs (2) all have zero flow. This situation is shown in fig. 5.2. □

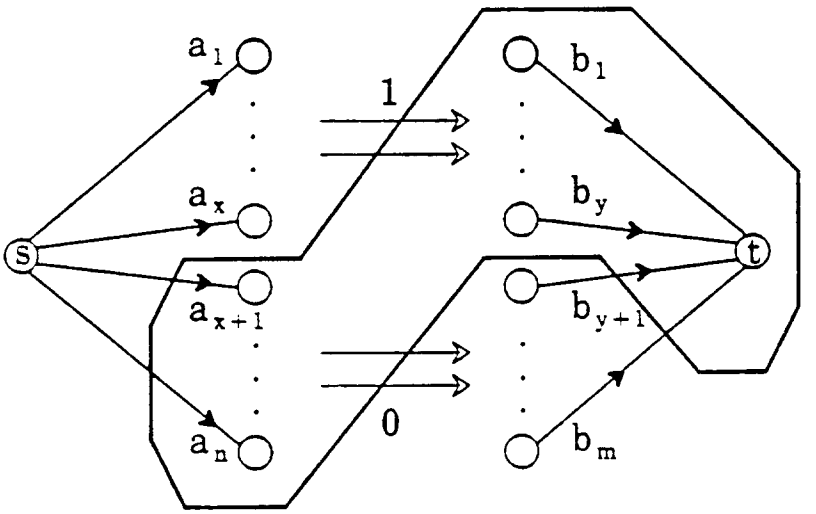

Fig. 5.2 : A tight cut - $sl_{\vec{a},\vec{b}}(x,y)=0$

All forward arcs are saturated; All backward arcs have flow 0

If $sl_{\vec{a},\vec{b}}(x,y) = 0$, We will call $C_{x,y}$ a *tight cut.* Proposition 5.2 shows that existence of a tight cut simplifies the solution considerably. In fact it gives rise to a divide and conquer approach: if $C_{x,y}$ is tight, constructing a matrix $M[1{:}n,1{:}m]$ for the original problem is reduced to constructing the two sub-matrices, $M[x+1{:}n,1{:}y]$ and $M[1{:}x,y+1{:}m]$. Of course, we are not always lucky enough to have a tight cut. Our approach is to **perturb** the input so as to improve our luck! Here is a high-level description of our algorithm:

(1) Perturb the inputs, $(\vec{a}, \vec{b})$. Call this new instance $(\vec{\alpha}, \vec{\beta})$.

(2) Recursively solve the instance $(\vec{\alpha}, \vec{\beta})$. Call the solution M'.

(3) Correct the matrix M' to obtain a matrix, M, which solves the original instance, $(\vec{a}, \vec{b})$.

How do we perturb an instance? A *basic perturbation* can be viewed as shifting one unit from the poor to the rich in order to make the situation tighter: subtract 1 from a_k and add 1 to a_l for some $k>l$. We do not allow that a perturbation will change the ordering of the a_i's, so it is necessary that $a_k > a_{k+1}$ and $a_l < a_{l-1}$ before the perturbation.

Remark: We will be discussing only perturbations of the row sums (the a_i's). All this discussion holds for perturbation of the column sums as well.

Proposition 5.3: Let $(\vec{a},\vec{b})$ be a problem instance, and let $(\vec{a},\vec{\beta})$ be obtained by shifting one unit from a_k to a_l for some $k>l$. Then $sl_{\vec{a},\vec{\beta}}(x,y) = sl_{\vec{a},\vec{b}}(x,y)-1$ if $l \le x < k$, and $sl_{\vec{a},\vec{\beta}}(x,y) = sl_{\vec{a},\vec{b}}(x,y)$ otherwise.

Proof: This can be seen by looking at the formula for sl. □

This proposition shows that a basic perturbation reduces the slack of a certain set of cuts, and leaves the rest unchanged. This observation is the basis for our algorithm.

5.2.2 One Phase of Perturbations

Achieving poly-log recursion depth for the basic algorithm described in the previous section is a non-trivial matter. The reason is that it is hard to control which cut or cuts will become tight. Furthermore, since we have limited ourselves to perturbations that do not change the ordering of the a_i's, it is not clear that a tight cut can always be obtained.

Say we are shifting units from a_k to a_l (for some $k>l$). How many units can we shift? Viewing the unit shifting as a sequential process (i.e. shifting one unit at each time step), we can shift until one of three things happens:

(1) a_l becomes equal to a_{l-1}.

(2) a_k becomes equal to a_{k+1}.

(3) $sl_{\vec{a},\vec{b}}(x,y)$ becomes zero, for some $l \le x < k$.

In case (3) progress is made, since a tight cut is created, and we can split the problem into two smaller problems. What about the first two cases? We observe that we have possibly *reduced the number of different a_i values*. This observation is the key to our approach for performing perturbations.

Definition: The *complexity of an instance* $(\vec{a},\vec{b})$ $comp(\vec{a},\vec{b})$ is the product of the number of different a_i values and the number of different b_j values.

Our parallel algorithm works in phases. The input to a perturbation phase is an instance of certain complexity, say K, and the output is one or more instances, each having complexity bounded by $c \cdot K$, for some constant $c<1$. Finally, if the complexity of the input is less than a certain constant, B, we construct a realization for it (this is the base case). We proceed to describe one perturbation phase. In

this discussion we will derive the constants c and B. For better exposition we will first describe a phase as a sequential process. The parallel implementation will be explained later.

In each phase either row sums or column sums are perturbed. The sequence that is perturbed (row or column sums) is that which has a larger number of different values. We will discuss a phase in which row sums are perturbed. Phases in which column sums are perturbed are essentially identical.

A phase starts by selecting a consecutive set of *active* rows, $\{h, h+1,...,l\}$. The parameters h and l depend on the input, $(\vec{a},\vec{b})$ and its complexity, K, and will be derived later. Let $L=a_{l+1}$ and $H=a_{h-1}$. The perturbation is performed as follows: repeatedly shift units from the lowest active row, (initially row l), to the highest active row, (initially row h). A row becomes inactive, and stops sending or receiving units, when its row sum either drops to L or reaches H. The phase terminates when one of two things happens:

(1) At most one active row is left.

(2) $sl_{\vec{a},\vec{b}}(x,y)$ becomes zero, for some $h \le x < l$.

In case (1) no tight cuts have been obtained, but the row sums of all the active rows (except, possibly, one) have become either L or H. Therefore the number of different row values decreases.

In case (2) one or more tight cuts are created, and the instance can be split, using proposition 2.2, into two smaller instances ("smaller", in this case, means less rows *and* lower complexity).

Let α,β and γ be the number of different values in the sets $\{a_1, \ldots, a_{h-1}\}$, $\{a_h, \ldots, a_l\}$ and $\{a_{l+1}, \ldots, a_n\}$ respectively. We want to select these parameters so as to minimize the complexity of the outputs of the phase:

Case (1) : The number of different row sums remaining is bounded by $\alpha+\gamma+1$ (since the β values corresponding to active rows disappeared, except for at most one).

Case (2) : Zero slack is obtained for one or more rows in the range $[h,l-1]$. A simple calculation shows that the number of different row sums in the resulting instances is bounded either by $\alpha+\beta+1$ or by $\beta+\gamma+1$.

Thus we need to minimize the maximum of $\alpha+\beta+1$, $\alpha+\gamma+1$ and $\beta+\gamma+1$ subject to $\alpha+\beta+\gamma=K$ (where $K = comp(\vec{a},\vec{b})$). The solution is, of course, to have α,β and γ as equal as possible, i.e. all roughly $K/3$. From this calculation one can see that the complexity can be reduced by these perturbations as long as the number of different row values is more than 5.

To summarize, if the input to a phase has complexity K, the outputs have complexity bounded by $\lceil\frac{2K}{3}\rceil+1$. Thus the total number of phases is $O(\log(n\cdot m))$. The base case is any instance with at most 5 different row values and 5 different column values.

We next discuss the parallel implementation of one perturbation phase. The first step is to calculate the new row sums and slack matrix under the assumption that none of the cuts become tight. If this new slack matrix is strictly positive then, indeed, we are in case (1).

Let p be the initial number of active rows ($p = l-h+1$). After the phase (assuming case (1)), there will be q rows of value H, $p-q+1$ rows of value L and one row of value I, where $H>I\geq L$. q and I are easy to calculate:

$$q = \lfloor\frac{\sum_{i=h}^{l}(a_i-L)}{H-L}\rfloor$$

$$I = \sum_{i=h}^{l}(a_i-L) \ \ mod \ \ (H-L)$$

Let $m_i = \min\{sl_{\vec{a},\vec{b}}(i,y) \mid 1\leq y\leq m\}$, and let m_i' be the new minimum slack in row i after the phase is completed (assuming case (1)). Then:

$$For \ \ h\leq i<h+q \qquad m_i' = m_i - \sum_{j=h}^{i}(H-a_j)$$

$$For \ \ h+q\leq i<l \qquad m_i' = m_i - \sum_{j=i+1}^{l}(a_j-L)$$

If all the m_i' are positive, then we are provably in case (1). If not, we need to detect at what "time step" (during the "sequential process") the first tight cut was created. This turns out to be a simple task for the following reason: if we plot the value of any entry in the slack matrix as a function of time, it decreases by one unit each step until some point in time, and remains constant from that point on.

Thus a row where the first zero slack occurs is a row for which m is minimum among the rows that have $m' \leq 0$. The total number of units shifted in the phase is this minimum m value. It is easy to compute the new row sums given the number of units shifted.

In both cases ((1) and (2)) we need to calculate the number of units shifted from row j to row i, for every $h \leq i < j \leq l$. (These numbers will be used later, in the correction phase.) This calculation can be performed by a simple partial-sums computation.

5.2.3 Correcting a Perturbed Solution

After a realization is obtained for the perturbed instance we need to correct it in order to obtain a realization for the original instance. Clearly the required task is to shift units back to their original rows. The rows which participate in the shifting of units are divided into two sets - the **donors** and the **receivers**, where donors shift units to the receivers during the perturbation phase, and get them back at the correction phase. Note that no row is both a donor and a receiver in any given phase. Let $s(j,i)$ be the the number of units shifted from the donor j to the receiver i in the perturbation phase.

Definition: Let M be a realization matrix. *Sliding* a unit from row i to row j means changing $M[i,k]$ from 1 to 0 and $M[j,k]$ from 0 to 1 , for some column, k.

Lemma 5.1: Given any realization of the perturbed instance, M', it is always possible to correct it by sliding $s(j,i)$ units from receiver, i, to donor, j, for all receivers and donors.

Proof: Again it is convenient to view the process of sliding units as a sequential one. Assume that some of the units have been slid, but less than $s(j,i)$ units have been slid from row i to row j. Call the current matrix M_1. We will show that it is possible to slide a unit from row i to row j in M_1, which proves the lemma.

Since units were shifted from row j to row i in the perturbation phase, it is the case that a_j was no larger than a_i before the phase began. Other perturbations in which rows i and j might have participated only increased the row sum of i and decreased the row sum of j. Now, since less than $s(j,i)$ units have been slid from

row i back to row j, it follows that row i has more 1's than row j in M_1. By the pigeonhole principle there is some column, k, such that $M_1[i,k]=1$ and $M_1[j,k]=0$.
[]

The implication of the proof above is that we do not need to be very careful in the way we slide units. The main problem we need to solve is that conflicts may arise when we slide many units in parallel. This could happen since a donor might have shifted units to many receivers, and a receiver might have received from many donors. Our goal is to break down the problem into a set of *independent* problems, which can all be solved in parallel. The first step is to get a formal description of the donor-receiver relation.

Definition: The *donation graph* $G=(D,R,E)$ is a bipartite graph with a vertex, $d_j \in D$, representing each donor and a vertex $r_i \in R$ representing each receiver, such that the edge $\{d_j,r_i\}$ is in E if and only if $s(j,i)>0$.

The following lemma plays a key role in simplifying the situation:

Lemma 5.2: The donation graph, G, is a forest.

Proof: Call a neighbor of a vertex, v, *nontrivial* if it has at least one other neighbor besides v. It follows from the way the perturbations were performed that each vertex, v, has at most two nontrivial neighbors, one that became inactive before v, and one that became inactive after v. Furthermore, all the vertices can be ordered according to when they became inactive. Therefore G cannot contain any cycles.
[]

One can see that a *matching* in the donation graph, G, corresponds to an independent set of sliding problems. However, there is no guarantee that the edges of G can be partitioned into a small set of matchings, since G might have vertices of high degree. Thus a more subtle partition is required.

Definition: A *constellation* is a subgraph of a given graph all of whose connected components are stars (where a *star* is a tree with at most one non-leaf vertex).

Lemma 5.3: The edges of a forest can be partitioned into two (edge-disjoint) constellations.

Proof: It suffices to show that the edges of a tree can be partitioned into two constellations. Let $T=(V,E)$ be a tree, and take it to be rooted at some vertex, P. The *level* of a vertex is its distance from P. v is the *parent* of u if $\{u,v\} \in E$ and v is closer to P than u. The partition of T into two constellations, $C_1=(V,E_1)$, $C_2=(V,E_2)$, is as follows:

$$E_1 = \{ \{u,v\} \mid u \text{ is the parent of } v, \text{ the level of } u \text{ is even } \}$$

$$E_2 = \{ \{u,v\} \mid u \text{ is the parent of } v, \text{ the level of } u \text{ is odd } \}$$

An example of such a partition is shown in fig. 5.3. []

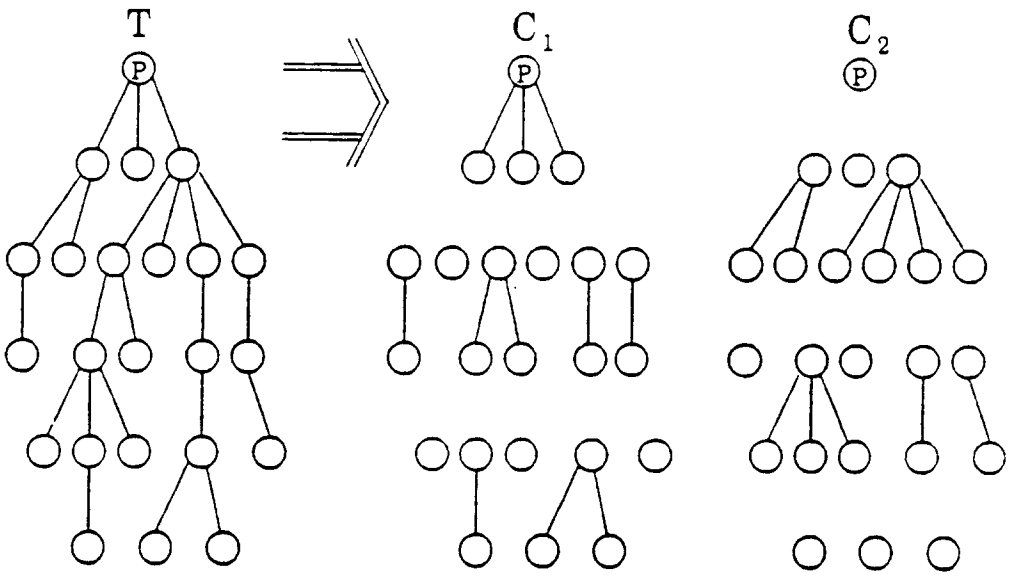

Fig 5.3 : Partitioning a tree into two constellations

Our solution is based on the observation that a constellation corresponds to a set of independent sliding problems which we can solve in parallel. Therefore our approach will be to partition the donation graph into two constellations and then to slide units in two stages - first corresponding to one constellation and then to the other.

A star in the donation graph corresponds to several donors with a common receiver or several receivers with a common donor. These two cases are symmetric, so we will discuss only the first one. In what follows we describe a parallel algorithm that slides all the units corresponding to a star with receiver R and donors $D_1, \ldots, D_d$. Let M be a realization matrix of the perturbed instance we are about to correct. Let r , $d_1, \ldots, d_d$ denote the number of 1's in rows R , $D_1, \ldots, D_d$ respectively and let $s_i = s(D_i,R)$. We need to slide s_i units from R to D_i, for all

$1 \le i \le d$ in parallel. Our approach is to solve a matching problem in the following bipartite graph, $B=(X,Y,E)$:

$$X = \{ x_j \mid M[R,j]=1 \}$$
$$Y = \{ y_{i,k} \mid 1 \le i \le d \ , \ 1 \le k \le s_i \}$$
$$E = \{ \{x_j, y_{i,k}\} \mid M[D_i,j]=0 \}$$

Lemma 5.4: Every matching of B which covers all the vertices in Y corresponds to sliding s_i units from R to D_i, for all $1 \le i \le d$ simultaneously.

Proof: By construction, there are $\sum_{i=1}^{d} s_i$ vertices in Y, one corresponding to each unit that was shifted from some D_i to R. There is an edge between x_j and $y_{i,k}$ if and only if a unit can be slid from row R to row D_i in column k. The claim is, therefore, evident. []

At first sight it seems that we need to solve a maximum bipartite matching problem, but closer observation reveals the following:

Lemma 5.5: Every *maximal* matching in B is maximum.

Proof: It suffices to show that any matching which does not cover all the vertices in Y can be extended. The degree of $y_{i,k}$ in B is, by definition, at least $r-d_i$. Before the perturbation phase the row sum of R was no less than that of row D_i. After the perturbations, the row sum of R increased by at least $\sum_{i=1}^{d} s_i$, and the row sum of D_i decreased by at least 1. Therefore:

$$\textit{For all } i,k \quad \textit{degree}(y_{i,k}) \ \ge \ r-d_i \ \ge \ \sum_{i=1}^{d} s_i + 1 \ = \ |Y| + 1$$

Since any matching contains no more than $|Y|$ edges it follows that no partial matching is maximal. □

A maximal matching can be constructed efficiently in parallel ([IS],[Luby]). Our parallel algorithm is, therefore, the following: construct the donation graph, and partition it into two edge-disjoint constellations, C_1 and C_2. For each component of C_1, construct the bipartite graph, B, as described, and find a maximal matching, F, in it. For all edges of B do in parallel: if $\{x_j, y_{i,k}\} \in F$ then slide a unit from R to D_i in column j. Finally, repeat this procedure on C_2 (with the updated

matrix).

It follows from lemmas 5.4 and 5.5 that after performing these operations all the perturbations (of the current phase) are corrected.

5.2.4 The Base Case

The base case for our algorithm is when the number of different values of row and column sums is bounded by a constant (5). The problem is then characterized by the different values: $a_1, \cdots, a_5$ and $b_1, \cdots, b_5$ and their multiplicities $n_1, \cdots, n_5$ and $m_1, \cdots, m_5$ respectively. Let M be the realization matrix we construct, and let M_{ij} be the submatrix of M induced on the rows with sum a_i and columns with sum b_j. We construct M in two steps:

Step 1: For each $i,j, 1 \leq i,j \leq 5$, determine the number, F_{ij}, of units in M_{ij}.

Step 2: For each $i,j, 1 \leq i,j \leq 5$, distribute the F_{ij} units between the different rows and columns of M_{ij}.

We carry out step 1 by constructing a flow network of constant size, and finding a max flow in it. The network has twelve vertices: a source s, a sink t, five "row" vertices $u_1, \cdots, u_5$, and five "column" vertices $v_1, \cdots, v_5$. The arcs are of three kinds: arcs from s to each u_i with capacities $n_i \cdot a_i$, from each v_j to t with capacities $m_j \cdot b_j$, and from each u_i to each v_j with capacities $n_i \cdot m_j$. This network is simply the result of taking the original network flow formulation for this problem, and compressing all "row" vertices with equal capacity into one vertex, and similarly for "column" vertices. Since this network is of constant size, a max flow can be constructed in constant time using standard sequential methods.

In step 2 we convert the solution for the compressed network to a solution for the original network by distributing the flow along each compressed arc evenly between the arcs it defines. We do this by providing a solution for the following problem: construct M_{ij} so that x_{ij} selected rows have each r_{ij} units, y_{ij} columns have each c_{ij} units and each of the remaining rows and columns have $r_{ij}-1$ and $c_{ij}-1$ units respectively. First, it is not hard to see that:

$$r_{ij} = \lceil \frac{F_{ij}}{n_i} \rceil \qquad x_{ij} = F_{ij} \bmod n_i$$

$$c_{i,j} = \lceil \frac{F_{i,j}}{m_j} \rceil \qquad y_{i,j} = F_{i,j} \bmod m_j$$

Assume we want each of the first $x_{i,j}$ rows and first $y_{i,j}$ columns to have $r_{i,j}$ and $c_{i,j}$ units respectively. Our solution is to put the units of the first row in the first $r_{i,j}$ columns, the units of the second row in the cyclically next set of columns etc. An example is shown in fig. 5.4.

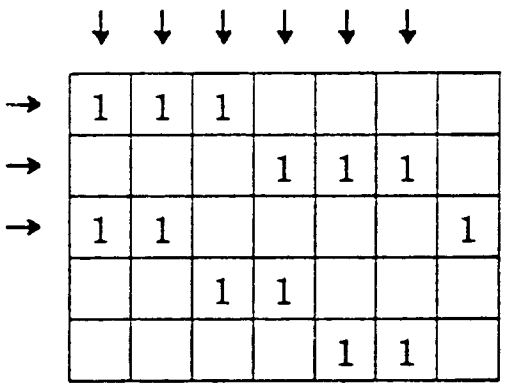

Fig. 5.4: Structure of $M_{i,j}$ with 5 rows, 7 columns and 13 units.

Selected rows and columns are marked with arrows.

A construction for arbitrary sets of selected rows and columns (not necessarily the first ones) is obtained from the one described above by simply permuting the rows and columns appropriately.

Now we are ready to construct a realization, M, for the base case. The values $F_{i,j}$ determine the $x_{i,j}$ and $y_{i,j}$ values. All we need to ensure is that any two rows (columns) with equal row (column) sums get selected the same number of times. This can be done by selecting the first $x_{i,1}$ rows in $M_{i,1}$, the cyclically next set of $x_{i,2}$ rows in $M_{i,2}$ and so on, and similarly for columns.

Since $\sum_{j=1}^{5} F_{i,j} = n_i \cdot a_i$, the total number of rows selected in $\{M_{i,1}, \ldots, M_{i,5}\}$ is an integer multiple of n_i, and it follows that any two rows with equal row sums are selected the same number of times. A similar argument holds for columns. Thus the construction described yields a correct solution for the base case.

5.2.5 The Algorithm

In this section we state the algorithm more formally. A few words about notation: I.P is shorthand for "in parallel". comments are between double parentheses; $l{:}k$ denotes a range of indices (in a matrix or a sequence); $\|$ denotes concatenation of sequences; $\#A$ is the cardinality of the set A.

procedure *MATRIX_CONSTRUCTION*$(\vec{a},\vec{b})$

((This is the recursive procedure for constructing a matrix, M, with given row sums, $\vec{a}$, and column sums, $\vec{b}$. The row and column sums are assumed to be given in a non-decreasing order.))

(1) Let n = length of $\vec{a}$; m = length of $\vec{b}$.

(2) Compute $V_{\vec{a}}$ and $V_{\vec{b}}$ - the number of different values in $\vec{a}$ and $\vec{b}$ resp.

(3) If $V_{\vec{a}} \leq 5$ and $V_{\vec{b}} \leq 5$ then return *BASE_CASE*$(\vec{a},\vec{b})$.

(4) $(\vec{\alpha},\vec{\beta},S,SL,pert,zerop)$ = *PERTURBATION*$(\vec{a},\vec{b})$.

(5) If **not** zerop then M' = *MATRIX_CONSTRUCTION*$(\vec{\alpha},\vec{\beta})$.

(6) Else let x,y be such that $SL[x,y]=0$ and either a_x is in the middle third of the $\vec{a}$ values or b_y is in the middle third of the $\vec{b}$ values. Do the following I.P:

(6.1) I.P set $M'[i,j]=1$ for all $1 \leq i \leq x$, $1 \leq j \leq y$.

(6.2) I.P set $M'[i,j]=0$ for all $x < i \leq n$, $y < j \leq m$.

(6.3)

$M'[x+1{:}n\ ,\ 1{:}y]$ = *MATRIX_CONSTRUCTION*$(\ \vec{\alpha}[x+1{:}n]\ ,\ \vec{\beta}[1{:}y]-x\ )$

(6.4)

$M'[1{:}x\ ,\ y+1{:}m]$ = *MATRIX_CONSTRUCTION*$(\ \vec{\alpha}[1{:}x]-y\ ,\ \vec{\beta}[y+1{:}m]\ )$

(7) M = *CORRECTION*$(M',S,pert)$.

(8) Return M.

end *MATRIX_CONSTRUCTION*

procedure *PERTURBATION*$(\vec{a},\vec{b})$

((This procedure computes one perturbation phase. The inputs are row sums, $\vec{a}$, and column sums, $\vec{b}$. The outputs are new row and column sums, $\vec{\alpha}$ and $\vec{\beta}$ resp, the

slack matrix SL, the matrix of numbers of units shifted S, a variable *pert* indicating whether row sums or column sums have been perturbed and a variable *zerop* indicating if zero slack is obtained.))

(1) Let n = length of $\vec{a}$; m = length of $\vec{b}$.

(2) Compute $V_{\vec{a}}$ and $V_{\vec{b}}$ - the number of different values in $\vec{a}$ and $\vec{b}$ resp.

If $V_{\vec{a}} \geq V_{\vec{b}}$ then set *pert* = "rows". Else set *pert* = "columns" and perform the rest of this routine with $\vec{b}$, $V_{\vec{b}}$ and m instead of $\vec{a}$, $V_{\vec{a}}$ and n resp.

(3) Find h and l for which $a_h \neq a_{h-1}$, $a_l \neq a_{l+1}$, and the number of different values in $<a_1, \ldots, a_{h-1}>$ and $<a_h, \ldots, a_l>$ are $\lfloor \frac{V_{\vec{a}}}{3} \rfloor$ and $\lceil \frac{V_{\vec{a}}}{3} \rceil$ resp. Let $H = a_{h-1}$ and $L = a_{l+1}$.

(4) Compute $q = \lfloor \frac{\sum_{i=h}^{l} (a_i - L)}{H - L} \rfloor$ and $I = \sum_{i=h}^{l} (a_i - L) \bmod (H - L)$.

(5) Compute $SL[i,j]$ ((the slack matrix)) for all $1 \leq i \leq n$, $1 \leq j \leq m$ I.P.

(6) Compute $m_i = \min \{SL[i,j] \mid 1 \leq j \leq m\}$ for all $h \leq i \leq l$ I.P.

(7) Compute $m_i' = m_i - \sum_{j=h}^{i} (H - a_j)$ for all $h \leq i < h + q$ I.P.

(8) Compute $m_i' = m_i - \sum_{j=i+1}^{l} (a_j - L)$ for all $h + q \leq i < l$ I.P.

(9) If $m_i' > 0$ for all $h \leq i < l$ then set $T = \sum_{i=h+q+1}^{l} (a_i - L) + \max \{0, a_{h+q} - I\}$.

Else set $T = \min \{m_i \mid m_i' \leq 0\}$, and set *zerop* to **true**.

(10) Initialize $S[i,j] = 0$ for all $1 \leq i,j \leq n$.

(11) $(\vec{a}', S) = SHIFT\_UNITS(<a_h, \ldots, a_l>, T, H, L)$.

(12) Set $\vec{a} = <a_1, \ldots, a_{h-1}> \,||\, \vec{a}' \,||\, <a_{l+1}, \ldots, a_n>$.

(13) Set $SL[i,j] = SL[i,j] - \sum_{k=h}^{i} \max\{0, a_k - a_k\}$ for all $h \leq i \leq l$, $1 \leq j \leq m$ I.P.

(16) Return $(\vec{a}, \vec{b}, S, SL, pert)$.

end *PERTURBATION*

procedure *SHIFT_UNITS*$(\vec{a}, T, H, L)$

((Shifts a total of T units between active rows with row sums $\vec{a}$. H is the upper bound on new rows sums and L is the lower bound. Returns the new row sums and the matrix, S, of the numbers of units shifted between pairs of rows.))

(1) Denote the elements of $\vec{a}$ by $a_h, \ldots, a_l$

(2) Compute for all $1 \le i \le T$ I.P:

$$d_i = \max \{ j \mid i \le \sum_{k=j}^{l} (a_k - L) \} \quad ((\textit{donor of unit } i))$$

$$r_i = \min \{ j \mid i \le \sum_{k=h}^{j} (H - a_k) \} \quad ((\textit{receiver of unit } i))$$

(3) Compute $S[i,j] = \#\{ k \mid d_k = i \, , \, r_k = j \}$ for all $h \le j < i \le l$ I.P.

(4) Compute $\alpha_i = a_i + r_i - d_i$ for all $h \le i \le l$ I.P.

(5) Return $(\vec{\alpha}, S)$.

end *SHIFT_UNITS*

procedure *CORRECTION*$(M,S,pert)$

((This procedure computes one correction phase. The inputs are a realization matrix, M, a matrix, S, containing amounts of units to be slid and a variable, *pert*, indicating if units need to be slid between rows or columns. The output is the matrix, M, after it has been corrected.))

(1) Let n = length of S.

(2) Construct the donation graph, G, where:

$$V(G) = \{1, \ldots, n\} \qquad E(G) = \{ \{i,j\} \mid S[i,j] > 0 \}$$

(3) For every connected component, T, of G do I.P:

(3.1) Partition T into two constellations, C_1 and C_2.

(3.2) Perform *SLIDE_UNITS*$(C,M,S,pert)$ for every connected component, C, of C_1 I.P.

(3.3) Perform *SLIDE_UNITS*$(C,M,S,pert)$ for every connected component, C, of C_2 I.P.

(3.4) Return M

end *CORRECTION*

procedure *SLIDE_UNITS*$(C,M,S,pert)$

((Units are slid in the matrix M, between one donor and many receivers or one receiver and many donors. The vertices of the star, C, are the participating rows/columns of M. The matrix, S, contains the numbers of units to be slid and the variable *pert* indicates if units need to be slid between rows or columns.))

(1) Let c be the unique non-leaf of C ((If C has exactly two vertices let c be any one of them)). Let $l_1, \ldots, l_d$ be the remaining vertices of C.

(2) If *pert* = "rows" then let $M_c, M_{l_1}, \ldots, M_{l_d}$ be rows $c, l_1, \ldots, l_d$ of M.

Else let $M_c, M_{l_1}, \ldots, M_{l_d}$ be columns $c, l_1, \ldots, l_d$ of M.

(3) If $S[c,l_1] > 0$ ((i.e. c is a donor and l_i are receivers)) then complement $M_c, M_{l_1}, \ldots, M_{l_d}$ I.P, and set *comp* to **true.**

Let $s_i = \max\{S[l_i,c], S[c,l_i]\}$ ((the number of units to be slid from M_c to M_{l_i})) for $1 \le i \le d$.

(4) Construct the bipartite graph, $B = (X,Y,E)$:

$$X = \{ x_j \mid M_c[j] = 1 \}$$
$$Y = \{ y_{i,k} \mid 1 \le i \le d \ , \ 1 \le k \le s_i \}$$
$$E = \{ \{x_j, y_{i,k}\} \mid M_{l_i}[j] = 0 \}$$

(5) Compute F, a maximal matching in B.

(6) For all $\{x_j, y_{i,k}\} \in F$ do in parallel: set $M_c[j] = 0$ and $M_{l_i}[j] = 1$.

(7) If *comp* then complement $M_c, M_{l_1}, \ldots, M_{l_d}$ I.P.

(8) Copy $M_c, M_{l_1}, \ldots, M_{l_d}$ back into their original location in M ((see step (2))).

end *SLIDE_UNITS*

procedure *BASE_CASE*$(\vec{a},\vec{b})$

((Constructs a matrix, M, with row sums $\vec{a}$ and column sums $\vec{b}$, where the number of different values of elements in $\vec{a}$ and $\vec{b}$ is at most five.))

(1) Let $a_1 > \cdots > a_k$ and $b_1 > \cdots > b_l$ be the values of the elements of $\vec{a}$ and $\vec{b}$ resp., and let $n_1, \ldots, n_k$ and $m_1, \ldots, m_l$ be their respective multiplicities.

(2) Construct a flow network, N, with vertices $s, t, u_1, \ldots, u_k, v_1, \ldots, v_l$ and the following arcs (for all $1 \le i \le k$, $1 \le j \le l$):

from s to u_i with capacity $n_i \cdot a_i$

from v_j to t with capacity $m_j \cdot b_j$

from u_i to v_j with capacity $n_i \cdot m_j$

(3) Find a max $s-t$ flow in N. For all i,j let $F_{i,j}$ be the flow on the arc (u_i,v_j).

(4) For all i,j construct $M_{i,j}$ as shown in figure 2.4. There are $F_{i,j} \bmod n_i$ selected rows, starting at row $(\sum_{h=1}^{j-1} F_{i,h}+1) \bmod n_i$ (cyclically) and $F_{i,j} \bmod m_j$ selected columns, starting at column $(\sum_{h=1}^{j-1} F_{i,h}+1) \bmod n_i$.

(5) Let M be the appropriate concatenation of the $M_{i,j}$'s.

(6) Return M.

end *BASE_CASE*

5.2.6 Parallel Complexity

The time and processor bounds of our algorithm depend on how we choose to implement the maximal matching routine. Two competing implementations are given in [IS] and [Luby]. On a graph with e edges, Israeli and Shiloach's algorithm takes time $O(\log^3 e)$ and uses $O(e)$ processors on a CRCW PRAM. Luby's algorithm requires only $O(\log^2 e)$ time on an EREW PRAM, but uses $O(e^2)$ processors. It is straightforward, though somewhat tedious, to verify that all the other operations in one phase of *MATRIX_CONSTRUCTION* can be performed with the resources required for maximal matching (in both the implementations listed above).

There are $O(\log|M|)$ phases, as proven in section 5.2.2 (Where $|M|=nm$). In a correction phase for rows there are $O(n)$ parallel calls to maximal matching on bipartite graphs with $O(m^2)$ edges each. When columns are corrected, there are $O(m)$ calls, each of size $O(n^2)$. Thus the number of processors required is $O(nm(n+m)) = O(|M| \cdot (n+m))$ using [IS], and $O(nm(n+m)^3) = O(|M| \cdot (n+m)^3)$ using [Luby]. When $n=\Theta(m)$ the processor requirements are $O(|M|^{1.5})$ and $O(|M|^{2.5})$ respectively.

5.3 The Symmetric Supply-Demand Problem

In this section we will show how the methodology developed in section 5.2 gives rise to a parallel algorithm to the symmetric problem. Here the input is a sequence of integers, $f_1 \geq f_2 \geq \cdots \geq f_n$, summing to zero. The goal is to construct a flow pattern in which every vertex can send up to one unit of flow to any other vertex such that the flow out of v_i minus the flow into it is f_i (for all $1 \leq i \leq n$). The goal can be viewed as constructing an $n \times n$ zero-one matrix, M (where $M[i,j]$ is the amount of flow sent from vertex i to vertex j) such that, for all i, the the number of ones in row i minus the number of ones in column i is f_i. Note that changing the values along the main diagonal does not change the instance M describes, so they can all be set to zero at the end of the computation.

Again we start with a network-flow formulation for the problem. The flow network has $n+2$ vertices: s, t, $v_1, \ldots, v_n$. If $f_i > 0$ then there is an arc from s to v_i with capacity f_i, and if $f_i < 0$ then there is an arc from v_i to t with capacity f_i. Also, there is an arc with capacity 1 from v_i to v_j for all $1 \leq i,j \leq n$. Examination of this network shows that there are only n potential min cuts: of all cuts containing x vertices with s, the one containing $v_1, \ldots, v_x$ is of smallest capacity. Thus, for this problem we have a *slack vector.* An analysis similar to the one in section 5.2.1 shows that, for all $1 \leq x \leq n$:

$$sl_f(x) = x \cdot (n-x) - \sum_{i=1}^{x} f_i$$

It is interesting to note that here, as opposed to the matrix construction problem, the object describing the slacks (a vector of length n) has a different size (and dimension) than the object being constructed (an $n \times n$ matrix).

A perturbation phase is performed in the same way as in section 5.2.2, except that there is only one sequence being perturbed (as opposed to separate row and column sequences). Again we have the property (similar to proposition 5.3) that shifting a unit from f_j to f_i ($i<j$) decreases the slacks at entries $i, i+1, \ldots, j-1$ by 1 and does not change the other entries.

A correction phase is, however, trickier than before. The reason is that if a unit is to be returned from entry i to entry j, it can be done either by sliding a unit from row i to row j or by sliding a unit from column j to column i. The

equivalent of lemma 5.1 holds here, but for each unit only one of the two ways of sliding listed above is guaranteed to exist. Furthermore, if we simultaneously try to slide units in rows and in columns, conflicts may arise (where a conflict is an attempt to slide two units into the same entry).

Our solution is to perform the correction in two stages: first slide between rows, then slide between columns. The first stage is identical to a row-correction phase of section 5.2. The only difference is that the maximal matching computed does not necessarily cover all the vertices of one side of the bipartite graph, B. After the first stage, we update the donation matrix (the $s(i,j)$'s), according to the numbers of units slid in the first stage. We then perform a column-correction phase for the resulting problem.

Lemma 5.6: Every maximal matching computed in the second stage is maximum.
Proof: As in section 5.2.3, let $R, D_1, \ldots, D_d$ be the vertices of a star in the donation graph. Let $B_1=(X_1,Y_1,E_1)$ be the bipartite graph for sliding between the rows corresponding to these vertices in the first stage. Let $B_2=(X_2,Y_2,E_2)$ be the bipartite graph for sliding between the columns corresponding to these vertices in the second stage. Then, as in the proof of lemma 5.5, for each vertex in Y_2, the sum of its degrees in B_1 and B_2 is at least $|Y_1|+1$. It follows that the degree of every such vertex in B_2 is at least $|Y_2|+1$. □

Corollary 5.1: Every unit that is perturbed gets slid in one of the two stages.

The base case is solved along the same lines described in section 5.2.4, but a few more details need to be handled. The base case is when there are at most five different values, $f_1 > \cdots > f_5$, with respective multiplicities $n_1, \ldots, n_5$. Again we start by finding a max flow in a constant size network (having 7 vertices - s, t, $v_1, \ldots, v_5$) to determine the number of units, $F_{i,j}$, in $M_{i,j}$. Now, as opposed to the previous case, $\sum_{j=1}^{5} F_{i,j}$ needn't be an integer multiple of n_i. Therefore, after distributing units evenly between all rows with the same f value (as described in section 5.2.4), some of these rows will have p units and some will have $p-1$ units (for some appropriate p). Similarly, not all the columns with the same f value will necessarily have the same number of units. We overcome this obstacle by

observing that if i and j have the same f value, and if row sum i is greater by one than row sum j then column sum i should be greater by one than column sum j. Therefore, the problem is solved by (using terminology of section 5.2.4) selecting rows and columns in the same order.

Finally we note that the algorithm for the symmetric problem uses the same resources (time and number of processors) as the matrix construction algorithm (see section 5.2.6).

5.4 Digraph Construction

In this section we describe our solution for the problem of constructing a simple digraph with specified in-degree and out-degree sequences. By "simple" we mean no self loops and no parallel arcs. Notice that if self loops are allowed, this problem is exactly the matrix construction problem described in section 5.2. The digraph construction problem can be stated as follows: given two equal-length sequences, $(o_1, \ldots, o_n)$ and $(i_1, \ldots, i_n)$, (that are not necessarily sorted!), construct an $n \times n$ zero-one matrix, M, that has o_k 1's in row k and i_k 1's in column k (for all $1 \leq k \leq n$), so that all the elements on the main diagonal of M are zero.

Our solution is based on the algorithm described in section 5.2. We start, again, by looking at the network flow formulation for this problem. The network is almost identical to the one in fig. 5.1, except that each vertex on the left is missing one outgoing arc, and each vertex on the right is missing one incoming arc. It is convenient to view the missing arcs as existing arcs with capacity zero. We will call these *blocked arcs* and the corresponding entries in the realization matrix *blocked entries*. Our first goal is to show that in this case too there are only n^2 potential minimum cuts. Let $a_1 \geq \cdots \geq a_n$ and $b_1 \geq \cdots \geq b_n$ be the sorted sequences of out-degrees and in-degrees respectively (i.e. $\vec{a}$ is obtained by sorting $\vec{o}$ and $\vec{b}$ by sorting $\vec{i}$), and let N be the network corresponding to $\vec{a}$ and $\vec{b}$ (similar to the one shown in fig. 5.1). The capacity of the cut $C_{x,y}$ (as shown in fig. 5.2) is, in this case:

$$capacity(C_{x,y}) = \sum_{i=x+1}^{n} a_i + \sum_{j=y+1}^{n} b_j + x \cdot y - B(x,y)$$

where $B(x,y)$ is the number of blocked arcs crossing the cut. Since there is at most

one blocked entry in every row and every column, a simple argument shows that if $a_x > a_{x+1}$ and $b_y > b_{y+1}$ then this cut has the smallest capacity among all cuts for which the s side contains x vertices on the left and $n-y$ vertices on the right. However, if, say, $a_x = a_{x+1}$ then a the cut obtained by switching vertices u_x and u_{x+1} might have smaller capacity, since the number of blocked arcs crossing it could be greater by one. Therefore, if we want the cuts $C_{x,y}$ to be the only potential minimum cuts, we need to be careful about the ordering of "row" vertices corresponding to rows with equal row sums, and similarly for columns. The conditions we need to enforce on the order are, simply: if $a_x = a_{x+1}$, then the blocked entry in row x should be in a lower-indexed column than the blocked entry in row $x+1$. The symmetrical conditions should hold for columns.

These conditions can be obtained by two rounds of sorting: first sort rows according to row sums. Sort rows with equal sums according to the corresponding column sums (i.e. the correspondence given by the $\vec{o}$ and $\vec{i}$ sequences), breaking ties arbitrarily. Now, sort the columns according to column sums. Columns with equal sums are sorted according to the order of the corresponding rows that was obtained in the first round. No ties can arise, since there is, at this point, a total ordering of the rows.

After this preprocessing is done, we are ready to proceed along the same lines as the algorithm described in section 5.2, with a few modifications. The slack function is now:

$$sl_{\vec{a},\vec{b}}(x,y) = \sum_{i=x+1}^{n} a_i - \sum_{j=1}^{y} b_j + x \cdot y - B(x,y)$$

By the discussion above, it is again true that an instance is realizable if and only if its slack matrix is non-negative. If $sl_{\vec{a},\vec{b}}(x,y)=0$ then $M[i,j]=1$ for all $1 \le i \le x$, $1 \le j \le y$ except for blocked entries, and $M[i,j]=0$ for all $x+1 \le i \le n$, $y+1 \le j \le n$.

The perturbation phases work identically here, since they only deal with the row and column sums, and not with the internal structure of the realization matrix.

In the correction phases there is a small modification - units should not be slid into blocked entries. This is fixed by modifying the bipartite graph, B, in the obvi-

ous way. Also, we need to re-examine the proof of lemma 5.5. It works out exactly right in this case, since it turns out that:

$$\textit{for all } i,k \qquad \textit{degree}(y_{i,k}) \geq |Y|$$

which is precisely sufficient (see the original proof).

The only tricky modification turns out to be for the base case. Again, there are at most five different row sum values and five different column sum values. The difficulty is that there are blocked entries scattered throughout. This spoils the simple cyclic realization that existed. We overcome this by partitioning the matrix into finer sub-matrices than in the previous case. Each of the $M_{i,j}$'s is partitioned further so that each sub-matrix either contains no blocked entries, or contains a blocked entry in every row and column.

Again we construct a realization in two steps. The first step is to determine the total number of units in each sub-matrix. This is done, here too, by solving a max flow problem (where the capacity of a sub-matrix is the number of non-blocked entries in it). Again, the network here is of constant size, so a max flow can be computed in constant time. In the second step, the units are distributed within the sub-matrices. The key here is to deal first with the sub-matrices containing blocked entries. It is not always possible to select arbitrary sets of rows and columns, but it is possible to distribute the units so that the discrepancy between any two rows or any two columns will be at most one unit. This can be done as follows: say the blocked entries are along the main diagonal (this will actually always be the case because of the preprocessing), and let k be the number of rows (and columns) of the sub-matrix. Let d_r (the rth diagonal) be the set of entries, (i,j), for which $j-i \equiv r \ (mod\ k)$. If F units are to be distributed, fill $d_1, \ldots, d_{\lfloor \frac{F}{k} \rfloor}$, and place the remaining units in $d_{\lfloor \frac{F}{k} \rfloor + 1}$. An example is shown in fig. 5.5.

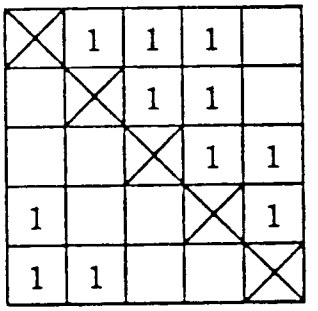

Fig. 5.5: A 5×5 sub-matrix with blocked entries containing 11 units.

Now, after the "problematic" sub-matrices have been dealt with, we can construct the sub-matrices with no blocked entries in the same fashion as described in section 5.2.4. The same arguments for proving validity of the scheme go through, because there is at most one blocked entry in every row or column.

5.5 Bounds on Supplies and Demands

Our parallel algorithm for the matrix construction problem can be extended to the case in which the sequences $\vec{a}$ and $\vec{b}$ represent upper bounds on row sums and lower bounds on column sums respectively. This is a natural extension of the matrix construction problem when rows represent supplies and columns represent demands.

Let $U = \sum_{i=1}^{n} a_i$ and $L = \sum_{i=1}^{m} b_i$. Let M be a realization matrix for the instance $(\vec{a},\vec{b})$, and let S be the number of 1's in M. Then, clearly, $L \leq S \leq U$. Say we fix S. Then the problem boils down to the following: modify the sequences $\vec{a}$ and $\vec{b}$ to obtain $\vec{\alpha}$ and $\vec{\beta}$ respectively so that:

(1) $\alpha_i \leq a_i$ and $b_j \leq \beta_j$ for all $1 \leq i \leq n$, $1 \leq j \leq m$.

(2) $\sum_{i=1}^{n} \alpha_i = \sum_{j=1}^{m} \beta_j = S$.

(3) $(\vec{\alpha},\vec{\beta})$ is realizable.

It is, of course, not always possible to satisfy all three conditions simultaneously. Thus our goal is find such a pair of sequences *if* it exists.

The key for obtaining the sequences $\vec{\alpha}$ and $\vec{\beta}$ is to consider the slack matrix,

as defined in section 2.1. Recall that the condition for realizability is that all the slacks are non-negative, and that:

$$sl_{\vec{\alpha},\vec{\beta}}(x,y) = \sum_{i=x+1}^{n} \alpha_i - \sum_{j=1}^{y} \beta_j + x \cdot y$$

where $\alpha_1 \geq \cdots \geq \alpha_n$ and $\beta_1 \geq \cdots \geq \beta_m$.

Lemma 5.7: Let $a_1 \geq \cdots \geq a_n$ and $b_1 \geq \cdots \geq b_m$. Let $\vec{\alpha}(k)$ ($\vec{\beta}(l)$) be the sequence obtained from $\vec{a}$ ($\vec{b}$) by subtracting 1 from a_k (adding 1 to b_l). Then $(\vec{\alpha}(1),\vec{\beta}(m))$ is realizable if $(\vec{\alpha}(k),\vec{\beta}(l))$ is (for any $1 \leq k \leq n$, $1 \leq l \leq m$).

Proof:

$$sl_{\vec{\alpha}(1),\vec{\beta}(m)}(x,y) - sl_{\vec{\alpha}(k),\vec{\beta}(l)}(x,y) = \sum_{i=x+1}^{n} (\alpha(1)_i - \alpha(k)_i) + \sum_{j=1}^{y} (\beta(l)_j - \beta(m)_j)$$

It is easy to see that for all values of x,y,k and l this difference is non-negative, which proves the lemma. □

Theorem 5.1: Let $\vec{\alpha}_{(S)}$ be obtained from $\vec{a}$ by repeatedly subtracting 1 from the largest element $U-S$ times and let $\vec{\beta}_{(S)}$ be obtained from $\vec{b}$ by repeatedly adding 1 to the smallest element $S-L$ times. Then $(\vec{\alpha}_{(S)},\vec{\beta}_{(S)})$ is realizable if there is any realizable pair of sequences $(\vec{\gamma},\vec{\delta})$ where $\gamma_i \leq a_i$, $\delta_j \geq b_j$ (for all $1 \leq i \leq n$, $1 \leq j \leq m$) and $\sum_{i=1}^{n} \gamma_i = \sum_{j=1}^{m} \delta_j = S$.

Proof: By induction on $U-S$ using lemma 5.7 □

$(\vec{\alpha}_{(S)},\vec{\beta}_{(S)})$ can be obtained from $(\vec{a},\vec{b})$ efficiently in parallel by a simple partial-sums computation. The algorithm is:

(1) For all S , $l \leq S \leq U$, do I.P:

(1.1) Compute $\vec{\alpha}_{(S)}$ and $\vec{\beta}_{(S)}$.

(1.2) Test if $(\vec{\alpha}_{(S)},\vec{\beta}_{(S)})$ is realizable ((using the method described in [FF])).

(2) Select an S for which $(\vec{\alpha}_{(S)},\vec{\beta}_{(S)})$ is realizable.

(3) Compute $M = MATRIX\_CONSTRUCTION(\vec{\alpha}_{(S)},\vec{\beta}_{(S)})$.

Steps (1.1) and (1.2) are simple partial-sum computations, and can be implemented using $O(n+m)$ processors. Since steps (1) and (2) can be implemented within the time and processor bounds used for step (3), the algorithm has the same

parallel complexity as the matrix construction algorithm. Note that we may perform step (2) with some criterion in mind (e.g. "construct a matrix with the smallest possible number of 1's subject to ...").

The extension of the symmetric supply-demand problem turns out to be even simpler. Here the natural extension would be that all the f values represent upper bounds, since making a number "less positive" corresponds to less supply, and making a number "more negative" corresponds to more demand. So in an instance of this problem, the positive numbers would sum up to $+H$ and the negative numbers would sum up to $-L$, for some $H > L$.

Here, as opposed to the matrix construction problem, it is clear which value of S works best (where S is the sum of the positive entries, and minus the sum of the negative entries). By looking at the expressions for the slack vector, one can see that decreasing S cannot ruin feasibility. Therefore S should be selected to be as small as possible, i.e. $S = L$.

To summerize, only the positive f entries should be modified. Again, as in the matrix construction problem, the best way to modify these numbers is to repeatedly subtract one unit from the largest entry until $H - L$ units have been subtracted.

References

[AHU] Aho, A.V. , Hopcroft, J.E. and Ullman, J.D. , "*The design and Analysis of Computer Algorithms*", Addison-Wesley, 1974.

[Atal] Atallah, M.J. , "Parallel Strong Orientation of an Undirected Graph", *Information Processing Letters*, 18, pp. 37-39, 1984.

[BT] Boesch, F. and Tindell, R. , "Robbins's Theorem for Mixed Graphs", *Amer. Math. Monthly*, 87, pp. 716-719, 1980.

[BW] Beineke, L.W. and Wilson, R.S. eds. , "*Selected Topics in Graph Theory*", Academic Press, 1978.

[Berge] Berge, C. , "*Graphs*", North Holland, 1985.

[CGT] Chung, F.R.K. , Garey, M.R. and Tarjan, R.E , "Strongly Connected Orientations of Mixed Multigraphs", *Networks*, 15, pp. 477-484, 1985.

[CL] Chartrand, G. and Lesniak, L. , "*Graphs & Digraphs" 2nd ed.* Wadsworth & Brooks/Cole , 1986.

[CW] Coppersmith, D. and Winograd, S. , "Matrix Multiplication via Arithmetic Progression", *Proc. 19th ACM Symp. on Theory of Computing*, pp. 1-6, 1987.

[Camion] Camion, P. , "Chemins et Circuits Hamiltoniens des Graphes Complets", *C.R Acad. Sci. Paris* (A) 249 pp. 2151-2152 , 1959.

[Cole] Cole, R. , "Parallel Merge Sort", *Proc 27th IEEE Symp. on Foundations of Comp. Sci.*, pp. 511-519, 1986.

[Cook1] Cook, S.A. , "Towards a Complexity Theory of Synchronous Parallel Computation", *L'Ensignment Mathematique* XXVII, pp. 99-124, 1981.

[Cook2] Cook, S.A. , "A Taxonomy of Problems with Fast Parallel Algorithms", *Information and Control*, 64, pp. 2-22, 1985.

[ET] Eswaran, K.P. and Tarjan, R.E. , "Augmentation Problems", *SIAM J. on Computing*, 5, pp. 653-665, 1976.

[FF] Ford, L.R. and Fulkerson, D.R. , "*Flows in Networks*", Princeton University Press, 1962.

[Fich] Fich, F. , "New bounds for Parallel Prefix Circuits", *Proc. 15th ACM Symp. on Theory of Computing*. pp. 100-109, 1983.

[GGKMRS] Gottlieb, A. , Grishman, R. , Kruskal, C.P. , McAuliffe, K.M. , Rudolph, L. and Snir, M. , "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", *IEEE Trans. Comput. C-32* 2 pp. 175-189, 1983.

[GJ] Garey, M.R. and Johnson, D.S. , "*Computers and Intractability*", W.H Freeman and Company , 1979.

[GP] Galil, Z. and Pan V. , "Improved Processor Bounds for Algebraic and Combinatorial Problems in RNC", *Proc. 26th IEEE Symp. on Foundations of Comp. Sci.*, pp. 490-495, 1985.

[GSS] Goldschlager, L.M. , Shaw, R.A. and Staples, J. , "The Maximum Flow Problem is Logspace Complete for P", *Theoretical Computer Science*, 21, pp. 105-111, 1982.

[GT] Goldberg, A. and Tarjan, R.E. , "A New Approach to the Maximum Flow Problem", *Proc. 18th ACM Symp. on Theory of Computing*, pp. 136-146, 1986.

[Gale] Gale, D. , "A Theorem on Flows in Networks", *Pacific J. Math.*, 7 , pp. 1073-1082, 1957.

[Gusf] Gusfield, D. , "Optimal Mixed Graph Augmentation", *Siam J. on Computing*, 16 (4), pp. 599-612, 1987.

[Hillis] Hillis, W.D. , "*The Connection Machine*" MIT Press , 1985.

[IS] Israeli, A. and Shiloach, Y. , "An Improved Parallel Algorithm for Maximal Matching in a Graph", *Information Processing Letters*, 22, pp. 57-60, 1986.

[KU] Karlin, A.R. and Upfal, E. , "Parallel Hashing - an Efficient Implementation of Shared Memory", *Proc. 18th ACM Symp. on Theory of Computing*, pp. 160-168, 1986.

[KUW1] Karp, R.M. , Upfal, E. and Wigderson, A. , "Are Search and Decision Problems Computationally Equivalent?", *Proc. 17th ACM Symp. on Theory of Computing*, pp. 464-475, 1985.

[KUW2] Karp, R.M. , Upfal, E. and Wigderson, A. , "Constructing a Perfect Matching is in Random NC", *Combinatorica*, 6 (1) , pp. 35-48, 1986.

[Ladner] Ladner, R.E. , "The Circuit Value Problem is log Space Complete for *P*", *SIGACT News*, 7,1, pp. 18-20, 1975.

[Lawler] Lawler, E.L. , "*Combinatorial Optimization, Networks and Matroids*", Holt, Reinhart and Winston, 1976.

[Luby] Luby, M. , "A Simple Parallel Algorithm for the Maximal Independent Set Problem", *Proc. 17th ACM Symp. on Theory of Computing*, pp. 1-10, 1985.

[MR] Miller, G.L. and Reif, J.H. , "Parallel Tree Contraction and its Application", *Proc. 26th IEEE Symp. Foundations of Comp. Sci.*, pp. 478-489, 1985.

[MVV] Mulmuley, K. , Vazirani, U.V. and Vazirani, V.V. , "Matching is as Easy as Matrix Inversion", *Proc. 19th ACM Symp. on Theory of Computing,*

[Moon] Moon, J.W. , "*Topics on Tournaments*", Holt, Reinhart & Winston , 1968.

[Naor] Naor, J. , "*Two Parallel Algorithms in Graph Theory*", Technical Report CS-86-6, Department of Computer Science, Hebrew University, June 1986.

[NS] Nisan, N. and Soroker, D. , "*Parallel Algorithms for Zero-One Supply-Demand Problems*", Report No. UCB/CSD 87/368, Computer Science Division, University of California, Berkeley, August 1987.

[PS] Papadimitriou, C.H. and Steiglitz, K. , "*Combinatorial Optimization: Algorithms and Complexity*", Prentice-Hall , 1982.

[Pipp] Pippenger, N. , "On simultaneous Resource Bounds", *Proc. 20th IEEE Symp. Foundations of Comp. Sci.*, pp. 307-311, 1979.

[RT] Rettberg, R. and Thomas, R. , "Contention is no Obstacle to Shared-Memory Multiprocessing" *CACM*, 29 (12) pp. 1202-1212, 1986.

[Redei] Redei, L. , "Ein Kombinatorischer Satz", *Acta Litt. Sci. Szeged*, 7 pp. 39-43 , 1934.

[Renade] Renade, A.G. , "How to Emulate Shared Memory", *Proc 28th IEEE Symp. on Foundations of Comp. Sci.*, pp. 185-194, 1987.

[Robbin] Robbins, H. , "A Theorem on Graphs with an Application to a Problem of Traffic Control", *Amer. Math. Monthly*, 46, pp. 281-283, 1939.

[Rober] Roberts, F.S. , "*Applied Combinatorics*", Prentice Hall , 1984.

[Ryser] Ryser, H.J. , "Traces of Matrices of Zeros and Ones", *Canad. J. Math.*, 9 ,

pp. 463-476 , 1960.

[SASLMW] Schneck, P.B. , Austin, D. , Squires, S.L. , Lehmann, J. , Mizell, D and Wallgren, K. , "Parallel Processor Programs in the Federal Government", *IEEE Computer*, 18 (6) pp. 43-56 , 1985.

[SV] Shiloach, Y. and Vishkin, U. , "An O(log n) Parallel Connectivity Algorithm", *J. of Algorithms*, 3, pp. 57-67, 1982.

[Soro1] Soroker, D. , "Fast Parallel Algorithms for Finding Hamiltonian Paths and Cycles in a Tournament", *J. of Algorithms*, to appear.

[Soro2] Soroker, D. , "Fast Parallel Strong Orientation of Mixed Graphs and Related Augmentation Problems", *J. of Algorithms*, to appear.

[Soro3] Soroker, D. , "*Optimal Parallel Construction of Prescribed Tournaments*", Report No. UCB/CSD 87/371, Computer Science Division, University of California, Berkeley, September 1987.

[TV] Tarjan, R.E. and Vishkin, U. , "An Efficient Parallel Biconnectivity Algorithm", *Siam J. on Computing*, 14 (4), pp. 862-874, 1985.

[Tsin] Tsin, Y.H. , "An Optimal Parallel Processor Bound in Strong Orientation of an Undirected Graph", *Information Processing Letters*, 20, pp. 143-146, 1985.

[Upfal] Upfal, E. , "A Probabilistic Relation Between Desirable and Feasible Models of Parallel Computation", *Proc. 16th ACM Symp. on Theory of Computing*, pp. 258-265, 1984.

[Vish1] Vishkin, U. , "*Synchronous Parallel Communication - a Survey*", TR 71, Dept. of Computer Science, Courant Institute, NYU, 1983.

[Vish2] Vishkin, U. , "On Efficient Parallel Strong Orientation", *Information Processing Letters*, 20, pp. 235-240, 1985.