PARALLEL ALGORITHMS FOR ZERO-ONE SUPPLY-DEMAND PROBLEMS

Noam Nisan[*] Danny Soroker[**]

Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

A technique which yields fast parallel algorithms for several zero-one supply-demand problems is presented. We give *NC* algorithms for the following related problems:
(1) 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.
(2) 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.
(3) Construct a digraph without self-loops with specified in- and out-degrees.

We extend our results to the case where the input represents upper bounds on supplies and lower bounds on demands.

1. Introduction

Supply-demand problems are fundamental in combinatorial optimization ([FF],[La]). 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 ([La],[PS],[GT]), but probably has no efficient parallel solution, since it is P-

* Research supported by NSF grant DCR-8411954 and a grant from Digital Equipment Corporation.
** Research 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.

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 paper 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 i'th row and b_j 1's in the j'th 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],[G]): 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.

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],[B]). 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 [B] 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.

We extend our results to the case where the input represents upper bounds on supplies and lower bounds on demands.

An outline of the paper follows:

In section 2 we explain in detail our methodology. We then state the matrix construction algorithm formally and finally discuss its parallel complexity (time and processor bounds).

In section 3 we describe how our techniques can be used to yield a solution to the symmetric supply-demand problem.

Section 4 contains a description of the algorithm for the digraph construction problem.

Finally, in section 5 we describe our method for solving the supply-demand problems when we are given upper bounds on supplies and lower bounds on demands.

A few words about parallel algorithms. Our algorithms use, in various places, partial sum computations. The basic problem in this category is - given a sequence $x_1, \ldots, x_n$, compute all sums of the form $\sum_{i=1}^{k} x_i$. This problem is widely mentioned in the literature (e.g. [F],[MR]) and we will not discuss it in this paper, other than mentioning that it can be solved efficiently in parallel. Several other tools that we use implicitly are finding connected components ([SV]) and various algorithms on trees ([TV]).

2. The Matrix Construction Problem

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 [R]. 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. 2.1: 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.

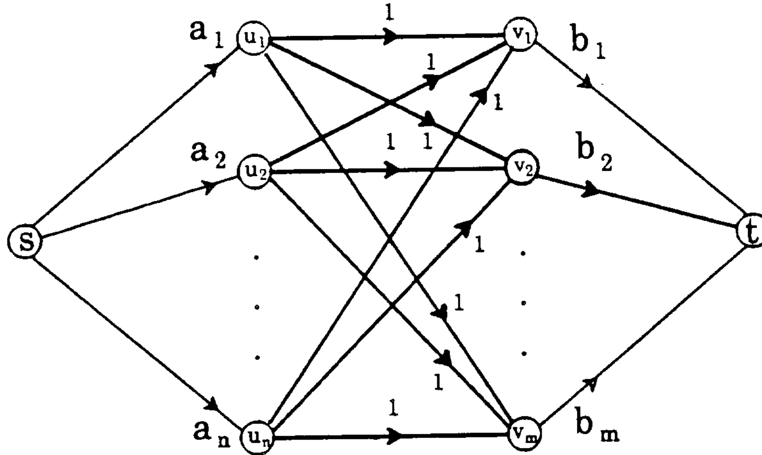

Fig. 2.1 : Flow network for solving the 0-1 matrix construction problem

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 s.t. $s \in C^s, t \in C^t$). 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, $\{C_{x,y} \mid 1 \leq x \leq n \ , \ 1 \leq y \leq m\}$, which are potential min cuts. The cut $C_{x,y}$ is shown in fig. 2.2. Therefore, necessary and sufficient conditions for the instance $(\vec{a},\vec{b})$ to be realizable are that for every $1 \leq x \leq n$ $1 \leq y \leq m$:

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

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

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

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,j'th entry is $sl_{\vec{a},\vec{b}}(i,j)$.

Proposition 2.1: The instance $(\vec{a},\vec{b})$ is realizable if and only if $SL_{\vec{a},\vec{b}}$ is non-negative.

Proposition 2.2: 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:

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

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

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. 2.2. □

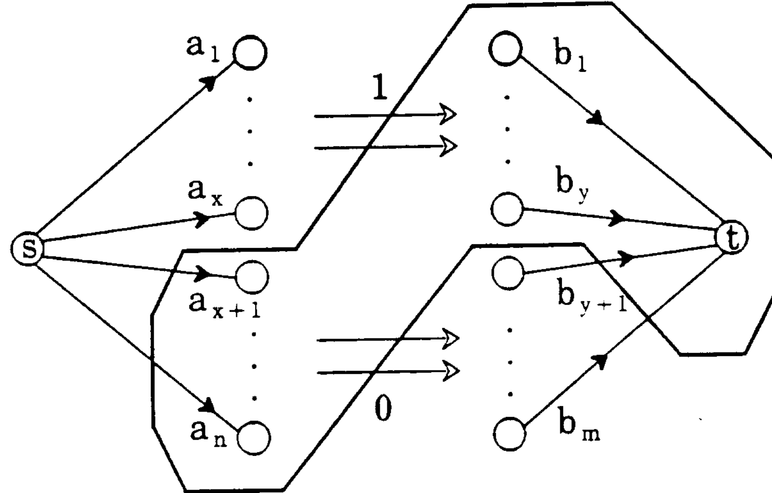

Fig. 2.2 : A tight cut - $sl_{\vec{a},\vec{b}}(x,y)=0$

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 2.2 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 crdering 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 2.3: Let $(\vec{a},\vec{b})$ be a problem instance, and let $(\vec{\alpha},\vec{\beta})$ be obtained by shifting one unit from a_k to a_l for some $k > l$. Then $sl_{\vec{\alpha},\vec{\beta}}(x,y) = sl_{\vec{a},\vec{b}}(x,y) - 1$ if $l \leq x < k$, and $sl_{\vec{\alpha},\vec{\beta}}(x,y) = sl_{\vec{a},\vec{b}}(x,y)$ 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.

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 $l \leq x < k$.

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:

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

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

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 the rows where the first zero slack occurs are the rows 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.

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 2.1: 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 2.2: 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 2.3: 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 v. The partition of T into two constellations, $C_1=(V,E_1)$, $C_2=(V,E_2)$, is as follows:

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

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

An example of such a partition is shown in fig. 2.3. []

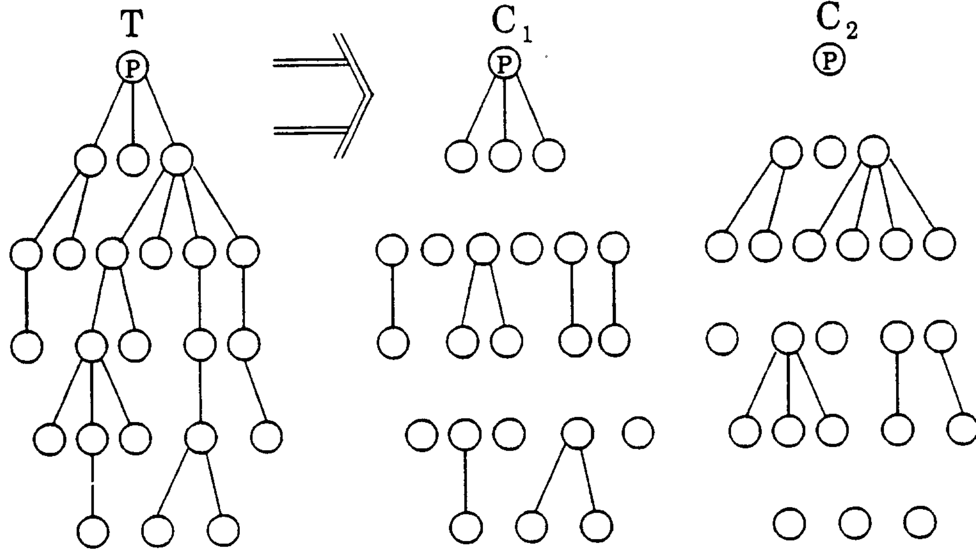

Fig 2.3 : Partitioning a tree into two constellations

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 2.4: 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 2.5: 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:

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

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],[Lu]). 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 2.4 and 2.5 that after performing these operations all the perturbations (of the current phase) are corrected.

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:

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

$$c_{ij} = \lceil \frac{F_{ij}}{m_j} \rceil \qquad y_{ij} = F_{ij} \ mod \ m_j$$

Assume we want each of the first x_{ij} rows and first y_{ij} columns to have r_{ij} and c_{ij} units respectively. Our solution is to put the units of the first row in the first r_{ij} columns, the units of the second row in the cyclically next set of columns etc. An example is shown in fig. 2.4.

	↓	↓	↓	↓	↓	↓	
→	1	1	1				
→				1	1	1	
→	1	1					1
			1	1			
					1	1	

Fig. 2.4: Structure of M_{ij} 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_{ij} determine the x_{ij} and y_{ij} 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.

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) $(\vec{\alpha}', S) = SHIFT_UNITS(<a_h, \ldots, a_l>, T, H, L)$.

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

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

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

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

(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 $(\vec{a}, S)$.

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 $1 \leq i \leq d$.

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

$$X = \{ x_j \mid M_c[j]=1 \}$$

$$Y = \{ y_{i,k} \mid 1 \leq i \leq d \ , \ 1 \leq k \leq s_i \}$$

$$E = \{ \{x_j, y_{i,k}\} \mid M_{l_i}[j]=0 \}$$

(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 $F_{i,j}$ *mod* n_i selected rows, starting at row $(\sum_{h=1}^{j-1} F_{i,h} + 1)$ *mod* n_i (cyclically) and $F_{i,j}$ *mod* m_j selected columns, starting at column $(\sum_{h=1}^{j-1} F_{i,h} + 1)$ *mod* n_i.

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

(6) Return M.

end *BASE_CASE*

2.6. Parallel Complexity

The time and processor bounds of our algorithm depend on how we chose to implement the maximal matching routine. Two competing implementations are given in [IS] and [Lu]. 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 2.2). 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 [Lu]. When $n=\Theta(m)$ the processor requirements are $O(|M|^{1.5})$ and $O(|M|^{2.5})$ respectively.

3. The Symmetric Supply-Demand Problem

In this section we will show how the methodology developed in section 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 <u>*slack vector.*</u> An analysis similar to the one in section 2 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 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 2.3) that shifting a unit from j to i ($i<j$) decreases the slacks at entries $i, i+1, \ldots, j-1$, 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 2.1 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.

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 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 3.1: Every maximal matching computed in the second stage is maximum.

Proof: As in section 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 2.5, 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 3.1: 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 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_{ij}, in M_{ij}. Now, as opposed to the previous case, $\sum_{j=1}^{5} F_{ij}$ needn't be an integer multiple of n_i. Therefore, after distributing unit evenly between all rows with the same f value (as described in section 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 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 2.6).

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 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 2. We start, again, by looking at the network flow formulation for this problem. The network is almost identical to the one in figure 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 figure 1). The capacity of the cut $C_{x,y}$ (as shown in figure 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 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 obvious way. Also, we need to re-examine the proof of lemma 2.5. It works out exactly right in this case, since it turns out that:

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

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 r'th 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 figure 4.1.

<table>
<tr><td>X</td><td>1</td><td>1</td><td>1</td><td></td></tr>
<tr><td></td><td>X</td><td>1</td><td>1</td><td></td></tr>
<tr><td></td><td></td><td>X</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td></td><td></td><td>X</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td></td><td></td><td>X</td></tr>
</table>

Figure 4.1: 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 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. 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 5.1: 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 5.1 □

$(\vec{\alpha}_{(S)},\vec{\beta}_{(S)})$ 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 problems, the positive number would sum up to $+H$ and the negative number 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.

Acknowledgments

We thank Richard Karp for suggesting the matrix construction and symmetric supply-demand problems and for interesting discussions.

References

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

[F] Fich, F. , "New bounds for Parallel Prefix Circuits"
15'th STOC, pp. 100-109, 1983.

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

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

[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"
18'th STOC, pp. 136-146, 1986.

[IS] Israeli, A. and Shiloach, Y. , "An Improved Parallel Algorithm for Maximal Matching in a Graph"
Manuscript, 1984.

[KUW1]
Karp, R.M. , Upfal, E. and Wigderson, A. "Are Search and Decision Problems Computationally Equivalent?"
17'th STOC, 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.

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

[Lu] Luby, M. "A Simple Parallel Algorithm for the Maximal Independent Set Problem"
17'th STOC, pp. 1-10, 1985.

[MR] Miller, G.L. and Reif, J.H. , "Parallel Tree Contraction and its Application"
26'th FOCS, pp. 478-489, 1985.

[MVV]
Mulmuley, K. , Vazirani, U.V. and Vazirani, V.V. , "Matching is as Easy as Matrix Inversion"
19'th STOC, 1987.

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

[R] Ryser, H.J. , "Traces of Matrices of Zeros and Ones"
Canad. J. Math. 9 , pp. 463-476 , 1960.

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

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