December 1984

Report No. STAN-CS-84-1028

Parallel Graph Algorithms

by

Peter H. Hochschild

Ernst W. Mayr

Alan R. Siegel

Department of Computer Science

Stanford University
Stanford, CA 94305

Stanford University
Department of Computer Science

Parallel Graph Algorithms

by

Peter H. Hochschild
Ernst W. Mayr
Alan R. Siegel

Support for this research included an NSF Graduate Fellowship and NSF grant MCS-82-03405 (PHH), an IBM Faculty Development Award (EWM), and NSF grant MCS-82-03405 and DARPA contract MDA-80-C-0107 (ARS)

TABLEOFCONTENTS

1. Introduction

This paper presents new paradigms to solve efficiently a variety of graph problems on parallel machines. These paradigms make it possible to discover and exploit the "parallelism" inherent in many classical graph problems. We abandon attempts to force sequential algorithms into parallel environments for such attempts usually result in transforming a good uniprocessor algorithm into a hopelessly greedy parallel algorithm. We show that by employing more local computation and mild redundance, a variety of problems can be solved in a resource- and time-efficient manner on a variety of architectures.

1.1 Parallel Machines

There is a great deal of literature concerned with solving graph problems on various kinds of parallel machines ([A-K], [CLC], [D-S], [H]). Our paradigms are applicable to both VLSI ([L-S],[U]) and distributed systems (as well as local networks, e.g. ethernets).

In our model of VLSI computation, the two most important parameters are area and time. Detailed features of VLSI technology will not concern us here. For our purposes, a VLSI circuit is both synchronous and digital. We make two assumptions of primary importance. First, we assume that every active component (e.g. gate, flip-flop, etc.) introduces a unit-time propagation delay. Second, we assume that every VLSI circuit is composed of at most some fixed number of layers, and that no two components on any given layer are separated by less than unit distance (unless they are connected together). This assumption implies that any region stores a quantity of information at most proportional to its area, and that the quantity of information crossing a boundary in unit time is at most proportional to the boundary length. Note that in our model, signal propagation time is independent of wire length. A formalization of this model can be found in [L-S].

1.2 Parallel Algorithms

To illustrate our techniques, we develop algorithms for solving such problems as connected components, minimum spanning forests, biconnected components and planarity testing. Our algorithms accept as input n-by-n matrices (adjacency matrices in the case of connected components, edge-weight matrices for minimum spanning forests). The matrices are read row by row. The successive rows are read in order, with a fixed time schedule. Thus the data is read once in a "when- and where-determinate" fashion (i.e. there is a fixed input schedule that determines when and where each input value is supplied). VLSI circuits for these algorithms require time and area of $O(n \log^c n)$ (for some small c), and

thus arc nearly optimal. Hereafter we will not write these powers of log n; instead we write f(n) = $O^*(g(n))$ if $f(n) = O(g(n)\log^c n)$ for some c.

Lipton and Valdes [L-V] present a circuit for computing connected components in area and time O*(n) but with a "when-indeterminate" I/O schedule (*i.e.* with data-dependent row read timing). We note that the obvious when-determinate implementations of the graph algorithms in [L-V] would run in $O^*(n^2)$ time.

In Chapter 2 we show how to remove this indeterminateness without sacrificing efficiency. Central to this task is the idea of filtration. A filter is a device used to discard irrelevant input data. This mechanism can reduce the storage, time, and communication requirements of a wide variety of problems. Filter construction demands balancing two opposing goals. On the one hand, a filter must operate quickly enough to avoid becoming a bottleneck. On the other hand, it must be thorough enough to discard a significant portion of the data. Thus, in general, a filter performs a kind of approximation to the desired computation. This approximation is later refined to yield the correct result.

In Chapter 3 we develop a more refined view of filtration and introduce the generally applicable concept of a *funnelled pipeline*. This concept is illustrated with algorithms for finding minimum spanning forests and biconnected components. Chapter 4 applies the funnelled-pipeline paradigm to planarity testing and related data-rearrangement problems. In Chapter 5 we present an alternative (and somewhat more powerful) algorithm structure, the two-stage funnelled pipeline.

In the last chapter WC explore a graph problem of greater parallel computational complexity, namely the strongly connected components problem. Using some of the funnelled-pipeline techniques, and a more liberal input schedule, we derive a relatively efficient algorithm. We also present lower bounds on the complexity of several related difficult graph problems.

2. Connected Components

2.1 Introduction

In this chapter we discuss the problem of finding the connected components of a graph. WC present a number of algorithms that solve this problem. Each of them demonstrates imp or tan t principles of const ructing efficient parallel algorithms. The progression of techniques leads to and motivates our notion of funnelled pipelines, the topic of the next chapter.

Definition 2.1: Given a graph G = (V, E) where $V = \{0, 1, \ldots, n-1\}$, define the function $cc : V \rightarrow V$ by

$$cc(j) = \min\{k \in V \mid k = j \text{ or } k \text{ is connected to j by a path in G}\}.$$

By the *connected components problem,* we mean the problem of computing the function cc from a graph G.

In all algorithms to follow, WC will assume that the graph is prcscntcd as an upper-triangular adjacency matrix. Furthermore, the matrix will be read row by row in a when-and where-determinate fashion.

The first problem to confront is the volume of input data. In order to achieve an O*(n) area circuit, most of that data must be discarded. In order for the circuit to bc time efficient, discarding must be rapid. We accomplish efficient data elimination by a Eltration process.

Jn the algorithms presented in this chapter, filtration will involve a benign form of deceit. We note that the exact edge structure of the input graph bears little relevance to its connected components; in fact there arc a variety of graph transformations that leave invariant the connected components. For example, suppose that the graph G contains a star-shaped subgraph consisting of a "hub" vortex c, "rim" vertices $v_1, v_2, \ldots, v_k$, and "spoke" edges $e_1, e_2, \ldots, e_k$ connecting c to $v_1, \ldots, v_k$ respectively. If we replace these spoke edges by a chain consisting of the edges $\{c, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$, we obtain a new graph that possesses exactly the same connected components as G. Indeed, any transformation that preserves path connectivity, preserves connected components.

2.2 A Divide-and-Conquer Connected Components Algorithm

Our first connected components algorithm is based on recursive deceitful filtration. We remark that in designing parallel "divide-and-conquer" algorithms, one must address not only the logical concerns vital to sequential recursion, but also the geometrical constraints imposed by data communication requirements. It is often tempting to shuffle data about; unfortunately this tends to waste a great deal of area or time.

The input to the algorithm is an n-by-n (upper-triangular) matrix *A*. For simplicity, assume that n is a power of two. Let the rows of *A* be denoted $A_0, \ldots, A_{n-1}$. WC may also assume that for each i, the i^{th} entry of the i^{th} row, denoted *A;(i),* is one. (If the input fails to obey this convention, it can be brought into compliance by the input stage.)

Taken faithfully, each row i represents a star. Vertex i forms the hub, the vertices $j > i$ for which *A;(j)* = 1 constitute the rim. However, as previously observed, it does no harm to pretend that these vertices are linked in a chain. Thus the algorithm will interpret each row as a statement indicating that the vertices whose corresponding entries are one should be linked together in a chain (proceeding from lowest to highest numbered vertex),

The algorithm operates in two major phases. The first phase consists of reading and filtering the adjacency matrix. At the conclusion of this phase, the circuit will have recorded a set of $O(n \log n)$ edges. The graph composed of these edges will have the same connected components as the original graph. This filtration reduces the number of edges from as much as $\Omega(n^2)$ to O(n log n); it is deceitful by virtue of the fact that it may store edges that were not originally present. The second phase merely computes the connected components induced by the stored edges.

The phase-one circuit is a recursive construction of filter units. We first describe the geometry and the interconnection of these filter units; their detailed operation is explained later. The topmost filter (which performs the outermost level of recursion) is of size n. It receives as input the rows of the adjacency matrix. Its output is fed to two filters of size $n/2$. Similarly, each of these units feeds two units of size $n/4$, etc. Each filter unit of size k is rectangular in shape with a row of k input ports along the top, and k output ports along the bottom. The recursion ends with filter units of size two; there are a total of n/2 of these. This arrangement is illustrated in Figure **2.1.**

A filter unit of size k accepts as input a sequence of bit vectors of length k. Each vector is processed and results in the output of a similar vector. Each output vector is cut into halves; one half is directed to each of the subfilters of size $k/2$. The rows of the adjacency matrix form the sequence of vectors input by the top filter. At all levels of recursion, vectors are interpreted in the same way; vertices corresponding to entries with value one are regarded as being connected to each other in a chain.

We now describe the function of the topmost filter unit. Let $L = \{0, \ldots, (n/2) - 1\}$ denote the first half of the vertices; let $R = \{n/2, \ldots, n-1\}$ denote the second half. The top filter is responsible for recording all information about connections joining vertices in L to vertices in R. This is its only job; connections between pairs of vertices in L and between pairs in R will bc handled recursively. Thus the graph is cut in half, information about interconnections between the halves is recorded, and finally, the halves are recursively

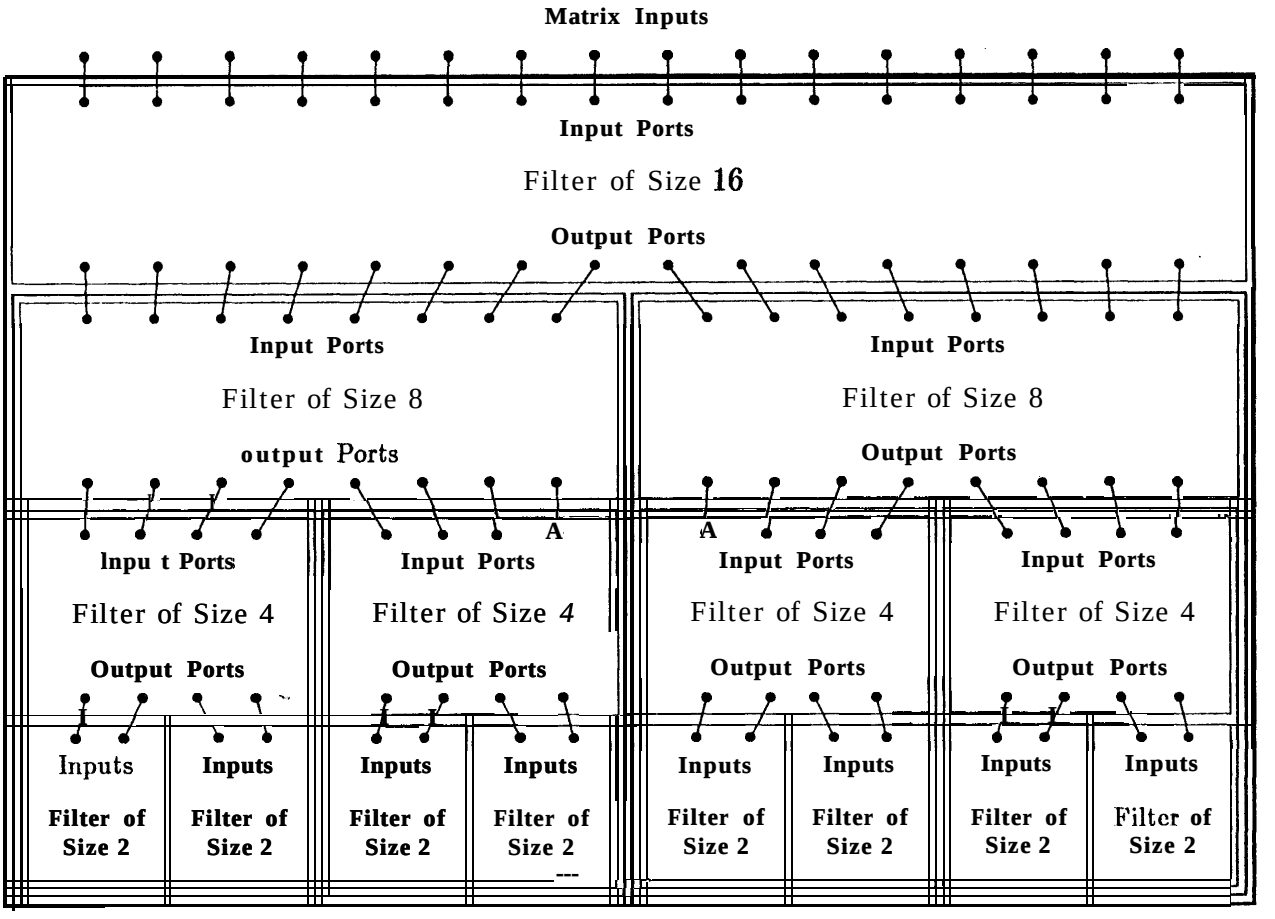

Figure 2.1: Divide-and-Conquer Architecture

and separately proccssed. Thus the filter separates the graph edges into three classes. E_L contains those edges with both endpoints in L, the set E_R contains those with both endpoints in II, and E_X contains the cross edges. E_L and E_R arc processed recursively; E_X is processed by the top filter.

The connection information derived from the edge set E_X is stored in an array $link(i)$ for $0 \leq i < n/2$. Each element $link(i)$ is associated with the vertex i (which is a member of L). The field *link(i)* contains either nothing or the name of a single vertex in R to Which the vcrtcx i is connected.

Initially all elements of *link* are empty. Whenever the top filter unit reads an input vector a, it first checks whether a contains both a one entry corresponding to a vertex in L and a one corresponding to a vcrtcx in R. If not, the vector implies no connection between L and R; the filter need merely transfer it to the output port for recursive processing. In other words, such vectors may contribute edges either to E_L or to E_R, but not to E_X. Otherwise let $l = \max\{j \in L \mid u(j) = 1\}$ and $r = \min\{j \in R \mid u(j) = 1\}$. l is the rightmost "active" vcrtcx in L while r is the leftmost active vcrtcx in R. The filter must now somehow record the fact that l is connected to r. No other connections from active

vertices in L to active vertices in R need be recorded; recall that the vector a is interpreted as a chain.

If the filter is lucky, $link(l)$ will be empty. In that case, the filter can simply set *link(Z)* to r and pass a on to the output for recursive processing; the edge $\{l, r\}$ is contributed to E_X, while the other edges in the chain represented by a are placed in E_L and E_R.

Otherwise the filter finds that $link(l)$ is already set to some vertex k in R. In this case the filter will set *u(k)* to one and pass this updated vector to the output. Thus the filter adds an edge to E_R instead of placing $\{l, r\}$ into E_X. That deceit is excused by the following observations:

- Connectivity between the active vertices in L and those in R is preserved by the fact that the filter has on record a connection from l to k and the fact that, recursively, k will be registered as being connected to r.
- Since l was connected both to all active nodes in R (by the current input vector a), and to node k (by some previous input vector), no new path connections are introduced by setting *a(k)* to one.

We now describe an implementation of the top filter unit. It consists of a binary tree of (simple) processors. There are n leaf processors; these are arranged, like the I/O ports, in a row. Each leaf processor i contains a one-bit cell *a(i)*. Each leaf processor i for $0 \leq i < \frac{n}{2}$ also contains the log n-bit cell *link(i)*. The interior node processors provide the ability to perform census functions (*e.g.* finding in $0^*(1)$ time the maximum of a set of numbers stored one per leaf); the entire unit is controlled by the root processor (refer to [L-V] for further detail).

The program executed by the root processor for a filter of size k is illustrated below:

```
procedure filter (k, n);
begin
  co graph size is n; filter size is k oc;
  for all i, 0 ≤ i < k/2 do in parallel link(i) := empty od;
  for timestep := 0 to n − 1 do
    co Read the input vector oc;
    for all i, 0 ≤ i < k do in parallel read a(i) from input port i od;
    co Check whether there is a connection between halves oc;
    if |{i | 0 ≤ i < k/2 and a(i) = 1}| > 0 and |{i | k/2 ≤ i < k and a(i) = 1}| > 0
      then
      l := max{i | 0 ≤ i < k/2 and a(i) = 1};
      r := min{i | k/2 ≤ i < k and a(i) = 1);
      if link(l) = empty then link(l) := r else a(link(l)) := 1 fi;
      fi;
    co Write out the possibly nltercd vector oc;
    for all i, 0 ≤ i < k do in parallel write a(i) to output port i od;
  od;
end filter .
```

Observe that the filter units of all sizes are similar (except that those of size two need produce no output vectors). Each filter unit of size k is of width $O^*(k)$, height $0^*(1)$ and requires time $0^*(1)$ per row of input. Each filter unit reads n vectors output by the

preceding filter (except the topmost, which reads the adjacency matrix); thus they can be pipelined so that the topmost reads one adjacency row every 0" (1) time units. The entire assembly of pipelined filters consumes O*(n) area and total time.

At the conclusion of the above described processing, a set of edges are stored in the *link* cells. This set of at most $\frac{n}{2}(\log n - 1)$ edges defines a graph with connectcd components identical to the original graph. The second phase, computing these connected components, can be accomplished in a variety of ways. The easiest, though not the most elegant, is to sequentially transfer the stored edges to a standard sequential processor executing a conventional connected coruponents algorithm (for example [AHU], [RND]). Because there are only $O(n \log n)$ edges, this costs only O*(n) area and time.

We observe that while the above divide-and-conquer approach is conceptually simple, it suffers a number of shortcomings. Prominent among these is its dependence upon rather specialized hardware. The next connected components algorithm we examine is based upon hardware of wider applicability.

2.3 A Tree-Machine Connected Components Algorithm

We develop a connected components algorithm that runs on a tree machine. This algorithm is in many respects similar to the previous algorithm. Indeed it is possible to simulate the divide-and-conquer algorithm on a single tree machine by a form of time sharing. However, we choose to implement a somewhat more symmetric algorithm whose structure remains more constant over varying values of n. (Recall the asymmetry in a divide-and-conquer filter of size *k:* only the leftmost *k/2* processors possess a *link* cell.)

WC begin with a tlcscriptiou of the structure and terminology of thc tree machine. Thc machine consists of a minimal depth binary tree having n leaves (placed as far left as possible), which arc called *Nodes.* The internal tree vertices nrc called *Switches.* Thus a Switch a has two subtrees, called a_{left} and a_{right}. Nodes i and j have a least common ancestor Switch, denoted by $lca(i, j)$.

The graph vertex j is represented in thc machine by loaf Node j. Node j can storc up to log n names of vertices belonging to thc same connectcd cotnponcnt as vertex j. The Switches will be used to communicate connection messages stating that two vertices are in the same component.

Definition 2.2: Let m be a Node and let s be its i^{th} ancestor Switch (*i.e.* m's first ancestor is its parent; its second ancestor is its grandparent, *etc.*) Then the set Of i^{th} *cousins* of m consists of the Nodes c such that $lca(m, c) = s$.

Note that the $\lceil \log_2 n \rceil$ or [log, n] sets of cousins of m form a partition of the other Nodes. 01)serve also that if vertex k is an i^{th} cousin of vertex j and Switch a is j's i^{th} ancestor, then either leaf Node k belongs to subtrcc a_{left} and j belongs to subtree a_{right}, or k belongs to a_{right} and j belongs to a_{left}. For example, if $n = 16$, Node 6 has the following sets of cousins: $\{7\}, \{4, 5\}, \{0, 1, 2, 3\}, \{8, 9, \ldots, 15\}$. Incidently, there is a close

relationship between the notion of cousins and the use of *link* fields in the divide-and-conquer algorithm.

The tree machine runs in big and *little* time steps. There are n big time steps, each composed of $\log^2 n$ little time steps. At the beginning of big step i, each leaf Node j reads bit $A_i(j)$ of the adjacency matrix. The little time steps are used to restructure the stored connection information to maintain the following invariant:

> At the completion of each big time step, leaf Node j stores the name of at most one vertex from each of its set of cousins. Furthermore, the vertices whose names are stored at Node j belong to the same connected component as j, and the connectivity relation defined by the stored edges is the same as that given by the adjacency matrix rows so far read.

When row i of the adjacency matrix is read, each Node j for $i + 1 \leq j < n$ is made "active" if $A_i(j) = 1$. Each active Node would like to store the vertex name i. However, an active Node may discover that it has already stored the name of another cousin from the set of cousins containing i. If that is not the case, Node j stores the name i. Otherwise, if j has already stored a competing cousin named h, where $h \neq i$, we say that j is a *contention* Node. In this case, Node j sends a message up the tree which says, (destination: i, message: *I am* already *connected* to *somebody* else, signed: j). We abbreviate the message by (i, j). The tree Switches route these messages along the tree to Node i; whenever two messages collide, one is arbitrarily discarded. Note that any message (i, j) received by Node i, will rise through the tree to the least common ancestor of Nodes j and i, whence it will descend to Node i. Node i processes only the last message it receives. This message comes from the least common ancestor of Node i and the contention Nodes. The victorious "maximally distant" message (i, m) arrives at Node i within 2 log n small time steps.

Notice that i will be in subtree $lca(i,m)_{left}$, and m will be in $lca(i, m)_{right}$. When Node i receives the message (i, *m),* it examines its storage location corresponding to the set of its cousins containing Node m. If that location is empty, it is set to m. Whether formerly empty or not, that storage location now holds some vertex named x. Node i now transmits the message (destination: $lca(i,m)$, tell *all active Nodes in your right subtree they may connect themselves to* x⟩ up the tree to $lca(i, m)$ (whence $lca(i,m)$ sends to all leaves in the subtree $lca(i, m)_{right}$ the message (connect yourself *to* x⟩.

At this point, the following facts hold:

- All contention Nodes are contained in the subtrec rooted at $lca(i,m)$.
- Node i has recorded the fact that i and x arc in the same component.
- x is in $lca(i,m)_{right}$, and i is in $lca(i,m)_{left}$.

In the next little time step, the contention Nodes in $lca(i, m)_{right}$ try to store vertex name x rather than i, and contention Nodes in $lca(i, m)_{left}$ try again to store i (and send messages to i since they fail). The connection scheme can run in parallel on both subtrees since no messages will reach $lca(i,$ m). This process must terminate after log n recursions (*i.e.* a total of $O(\log^2 n)$ little time steps).

A big time step is now completed, and adjacency matrix row i+l can be read. After all rows have been read, the remaining connected components problem can be satisfactorily solved by any standard sequential algorithm. Morcovcr, it is easy to devise reasonable

algorithms that utilize the existing machine structure. The algorithm in [L-V], for example, suffices.

We note that with simple modifications the algorithm can be pipelined (by traditional means) to run in $O(n \log n)$ little time steps (instead of $O(n \log^2 n)$ little time steps). That performance improvement can be obtained at the cost of only a few registers per node; no other machine structures are needed. This simplicity is due in part to the computational power of a tree of internal Switches. Such a structure has other uses. It can, for example, link an arbitrary subset into a linear list in $\log n$ time. Moreover, a circular shift (of one step) can be implemented for such a subset list in $O(\log n)$ time.

We also briefly remark on another possible implementation of the pipelined tree-machine algorithm. Instead of providing $\log n$ storage cells at each Node, one can provide storage at the Switches. In particular, if C is one of the sets of cousins of a Node j, we place the corresponding storage cell not at Node j, but at the Switch associated with the least common ancestor of the set {j} U C. Thus there are n storage cells located at the root Switch, $n/2$ at each of its two children, etc. This provides an alternative structure for implementing an $O^*(n)$-area, $O(n$ log n)-little-time-step pipelined algorithm.

The tree-machine filter makes heavy use of the tree structure. The latter stages of the divide-and-conquer scheme rely on multiple simultaneous switching in the interconnection tree; the completion of a big time' step may require sending $\Omega(n)$ messages. Thus the entire process may involve $\Omega(n^2)$ messages. A simulation by a system of n processors interconnected by an ethernet [M-B] or a bus (a one-word concurrent-read-prioritized-write PRAM [IT-W]) would therefore require $\Omega(n^2)$ time. This observation raises the question of whether the amount of communication can be reduced sufficiently to permit an efficient ethernet algorithm. In the following section we find that the answer, perhaps surprisingly, is yes.

2.4 Computing Connected Components in Low-Communication Environments

We now present a parallel algorithm for computing connected components which requires only minimal interprocessor communication. As before, this "ethernet" algorithm receives its input in the form of an adjacency matrix, is when- and where-determinate, and requires time 0* (n). If implemented in VLSI, it consumes O*(n) area. If implemented as a distributed system, n processors, equipped with $O(\log n)$ memory words and interconnected by an ethernet or bus, suffice.

We assume that the architecture consists of n conventional processors. The processors are interconnected sufficiently well to implement the following operations in at most O*(l) time.

- Broadcast: Any processor may execute this function to broadcast a message of length $O(\log n)$ to all other processors. At most one processor may be engaged in the execution of this command at any time.

- *Minimum*: Suppose that each processor i has an $O(\log n)$-bit register *priority; and a* one bit register $awake_i$. Execution of the Minimum operation simultaneously informs all processors of a value m (if any) such that $priority_m = \min\{priority_i \mid awake_i = 1\}$.
- *Maximum*: Execution of this operation simultaneously informs all processors of a value m (if any) such that $priority_m = \max\{priority_i \mid awake_i = 1\}$.

For purposes of exposition, we will assume that the *Maximum* and *Minimum* functions are provided as primitive operations. Of course, in practice, they would be implemented by subroutines using only primitive communication functions. Note that the *Maximum* and Minimum operations can be accomplished in $O(\log n)$ time with a binary search algorithm executing on a bus or ethernet architecture with collision detection. (The collision detection provides nothing more than the n-way "OR" function.) *Maximum* and Minimum operations can be used in order to designate a unique processor for the Broadcast operation.

As before, each vertex i is represented by processor i. Each processor i contains an array $link_i$, a counter $count_i$, and a few miscellaneous temporary registers and flags. (All registers are of length $\lceil \log n \rceil$.) Each entry $link_i(j)$ for $0 \leq j < count_i$ can be understood to represent the edge $\{i, link;(j)\}$. The value of $count_i$ is thus the number of edges in processor i's *link* list.- For simplicity of exposition, we will initialize *link;* with the single entry i.

At every time for every vertex i, the list consisting of $link_i(j)$ for $0 \leq j < count_i$ contains names of vertices to which i is known to bc connected. The entries of this list will always be distinct and sorted, that is, $link;(j) < link_i(k)$ for $0 \leq j < k < count_i$. It may help the reader to think of the smallest entry, $link_i(0)$, as vertex i's current estimate of its connected component number $cc(i)$.

The main difficulty is to keep the *link* lists from growing too large. This goal is accomplished by what can best be described as an abasing process. From time to time, a uniquely designated processor, say processor i, will initintc an abasing event. This event will result in giving all names stored in $link_i$ a single new alias b. This is accomplished by having processor i broadcast its list $link_i$. All processors, including processor i, will monitor this activity; any processor storing a name that was in $link_i$, deletes it, and instead remembers the name b. In this way, the new alias, b, is made universal; the old aliases arc forgotten everywhere and forever. It may bc that some processor's *link* list had contained several of the old names; in that case the list becomes shorter. Indeed, the point of performing aliasing operations is to shorten *link* lists. The key to choosing which vertices to alias, therefore, is to pick ones that arc so "popular" that rc-abasing them causes lists to contract significantly.

For convenience wc let, as before, the adjacency matrix A;(j) be upper triangular with ones along the main diagonal. The algorithm runs in n *big* time steps, each composed of $O(\log n)$ *little* time steps.

At each big time step i, each processor j reads $A_i(j)$ and sets $awake_j = A;(j)$. (Recall that $awake_i$ will therefore bc set to one.) All processors j set the temporary register cc_j equal to $link_i(0)$. The *Maximum* operation is now used to select an awake processor k with largest count value. Vertex k will play the role of being "most popular".

Next, processor k iteratively broadcasts each of its *link* entries. Among the values transmitted will be cc_k. During the broadcasts, all processors listen to the transmitted values. Any processor storing such a value in its *link* array, deletes it and becomes (or stays) awake. At the conclusion of the broadcasts, $count_k$ will be zero, k will still be awake, and cc_k will still be recorded. Note that all processors awakened by a one in A; remain awake.

Now a minimum cc value (*i.e.* alias) is selected from the set of awake processors and broadcast. All awake processors store that alias in their link array. Note that this alias will be the smallest name recorded in any awake processor. All names that were broadcast by the most popular processor k have been globally aliased. Moreover, all processors j for which A;(j) = 1 have recorded links to the possibly new alias of vertex k. Thus the connectivity information represented by the adjacency matrix row has been recorded. A big time step has now been completed.

Our algorithm is presented below. We assume for the moment the existence of the insertion and deletion subroutines used to manipulate the *link* lists. These routines are assumed to maintain sorted order and to update the *count* registers in the appropriate manner.

```
program components(n);
begin
  co Initialization oc;
  for all j, 0 ≤ j < n do in parallel
    count_j := 0 od;
    insert j into link_j
    od;
  co Filtration oc;
  for i := 0 to n − 1 do
    co Start a big time step oc;
    co Read adjacency matrix row oc;
    for all j, 0 ≤ j < n do in parallel
      awake_j := A_i(j);
      cc_j := link_j(0)
      od;
    co Sclcct the most popular awake processor oc;
    let k be so that awake_k = 1 and count_k = max{count_j | 0 ≤ j < n and awake_j = 1};
    co Broadcast and delete k's list oc;
    for all v ∈ link_k do
      Broadcast v;
      for all j, 0 ≤ j < n do in parallel
        if v is a member of link_j then
          Delete v from link_j;
          awake_j := 1
          A
        od;
      od;
    co note that count_k = 0 oc;
```

co Pick the new alias a oc;
$a := \min\{cc_j \mid 0 \le j < n$ **and** $awake_j = 1\}$;
co Re-alias all awake vertices – make a their new alias oc;
for all j, $0 \le j < n$ **do in parallel**
if $awake_j = 1$ **then** Insert a into $link_j$ **fi**
co Note that since a is smallest, a is always stored in $link_j(0)$. Thus, during next big time step, cc_j will be equal to a. oc;
od
co End of big time step oc;
od;
end *connected.*

Our next task is to examine how long each big time step takes. Only one loop causes much concern, namely that in which the most popular processor's *link* entries are iteratively broadcast and deleted from all lists. All other operations performed during a big time step require at most O*(l) time. The time required for the iterative broadcast and delete loop depends on the length of the link arrays. Let *bound* denote the maximum value ever attained by any count register. We observe that since the *link* arrays are sorted, the loop can be done in a number of steps bounded by $O(bound(\log^c n + \log bound) + bound)$ even under the most pessimistic architectural assumptions. This formula charges $\log^c n$ for each *Broadcast*, log *bound* to check whether the broadcast value appears in a list (if so, the entry is flagged), and *bound* for each processor to delete the flagged entries and compact the list. There arc, of course, a variety of other efficient ways to implement the loop; the number of steps can be reduced to $O(bound)$ at the expense of providing each processor with an additional buffer array of length *bound*. We will show later that *bound* never exceeds $\lceil \log n \rceil + 1$.

Two issues remain to be addressed. First we must establish correctness of the filter; it must be shown that no connected component information is lost. Second, we must verify that $bound \le \lceil \log n \rceil + 1$. This upper bound will guarantee a total running time (and area) of $O^*(n)$.

Correctness of the filter is established by the following invariant.

> **At** the completion of each big time step i, the edges rcprcscnted by the *link* entries induce the same connected components as the edges comprising the first i adjacency matrix rows.

This is easily verified by induction.,

The proof of the running time is somewhat technical; the trusting reader may wish to skip over it. In order to establish a bound on the length of the *link* arrays, we must examine the conditions under which such a list grows. We make the following observations.

- During each big time step i, only those processors j for which $A_i(j) = 1$ can increase the length of their lists. Furthermore, the length of each such list can grow by at most one.
- If during some big time step the list $link_j$ grows, it must have had no more entries at the beginning of the time step than did the processor designated "most popular" whose

list was broadcast. We say that the most popular processor "shielded" processor j. Furthermore, $link_j$ must have had no entries in common with the broadcast list.

- All stored edges point to the left; that is, at all times, for all j and *k*, we have $link_j(k) \leq j$.
- Suppose that at some big time step the most popular processor, *k*, broadcasts its list $\{l_p \mid l_p = link_k(p)$ for $0 \leq p < count_k\}$. Then, for all subsequent time, no name in $\{l_p \mid p \geq 1\}$ appears in any *link* list. In other words, once a vertex is re-aliased, its old alias becomes extinct. Incidentally, the name l_0 may or may not become extinct, depending on whether it is selected as the new alias.

Armed with these observations, we make the following definitions.

Definition 2.3: A growth event *of* order p, denoted g(j, $t, p,$ a), occurs whenever a big time step t results in increasing the value of $count_j$ from $p - 1$ to p by installing the new alias a into $link_j$. (Growth events of order one are denoted $g(j, -1, 1,$ j); these occur during initialization.)

Definition 2.4: **An** *aliasing* **event of** *order* **p, denoted** $m(k, t, p, l)$, occurs whenever during big time step *t* the most popular processor *k* broadcasts the *p* names $link_k$ *(0), link&), ..., $link_k$ (p* **— 1).** The list 1 contains the names thereby made extinct. Thus l equals $\{link_k(i) \mid 0 < i < p\}$ or $\{link_k(i) \mid 0 \leq i <$ p} according to whether $link_k(0)$ was chosen as the new alias.

We note that every growth event g(j, $t,$ p, a) is naturally associated with a ***shielding aliasing*** event of order $p' \geq p - 1,$ namely the simultaneously occurring aliasing event $m(k, t, p', l)$ in which the most popular processor *k* shielded j. It is also naturally associated with a precursor *growth* event g(j, t', p **— 1,** a) (for the sake of definiteness, choose the most recent if there were several) in which $count_j$ attained the value $p - 1.$

Similarly, each aliasing event $m(k, t, p, l)$ has a naturally associated ***precursor*** *growth event* of order *p,* namely the growth event *g(k, $t', p,$ a)* most recently cxpcricnccd by processor *k.*

WC now describe a *critical* ***event*** *tree.* It is **a** recursively-defined binary tree each of whose nodes is labeled with a growth cvcnt. If a node has as its label a growth event of order one, then the node is a leaf. Otherwise suppose that a node *n* has label g. Then the left child of *n* is labeled with g's precursor growth event and the right child of n is labeled with the precursor of the shielding aliasing event associated with g.

We say that a critical event tree is of order *p* if the label of its root is of order *p.* By the above definition, we see that a critical event tree of order p contains at least 2^{p-1} leaves. Recall that each leaf has a label of the form g(j, — 1, 1, j); WC call j the leaf **name.** What remains to be shown is that all of these leaf names arc distinct. This fact will assure that no tree has order greater than $\lfloor \log n \rfloor + 1.$

Definition 2.5: Let alias(v, t) denote **the alias of vertex v at the end of big** time step $t.$ (Let alias$(v, -1)$ = v for all v.)

That the above definition is meaningful is a consequence of the fact that all aliasing operations are done globally. Thus, if $m(k, t, p, l)$ is an aliasing event in which a is the new alias chosen, then for all v we have that

$$\text{alias(v, } t) = \begin{cases} \text{alias}(v, t-1), & \text{if } link_i \cap l = \emptyset; \\ a, & \text{if } link_v \cap l \neq \emptyset. \end{cases}$$

In particular, if $t' > t$ then alias(alias(v, t), t') = alias(v, t').

Lemma 2.1: If at some time t, the list $link_j$ contains the name v, then for all times $t' > t$, the name $\text{alias}(v, t')$ is a member of $link_j$.

Proof 2.1: Whenever a name is deleted from a list, its new alias is inserted into the list during the same time step. ∎

This lemma establishes that once a name joins a *link* list. it is forever represented in that list by an alias.

Lemma 2.2: Suppose that g = g(j, $t, p,$ a) is a growth event and that the leaf named v is one of its descendants. Then at the end of time step t (and forever after), we have $\text{alias}(v, t)$ in $link_j$.

Proof 2.2: This is established by induction. Suppose that v is in the left subtree of g and that $g(j, t', p-1, a')$ is the left child of g. By induction, $\text{alias}(v, t')$ appeared in $link_j$ at time t', hence since $t' < t$, we have $\text{alias}(v, t)$ in $link_j$ at time t as required. If v is in the right subtree of g and $g(k, t', p', a)$ is the right child of g, we have by induction that at the end of time t', the name $\text{alias}(v, t')$ is in $link_k$. But then, since k was most popular at step t, we have that $\text{alias}(v, t)$ = a. Hence $\text{alias}(v, t)$ is in $link_j$ at the end of time t. ∎

Theorem 2.1: The leaf names appearing in a critical event tree are all distinct.

Proof 2.3: The proof follows by a simple induction. Let g = g(j, t, p, a) be the root of a critical event tree. Let L be the set of leaf names in the left subtree; let R be the leaf names in the right subtree. We show that $L \cap R = \emptyset$. Suppose that v $\in L \cap R$. Let g($j, t', p-1, a'$) and $g(k, t'', p'', a'')$ be the left and right children of g respectively. Now at the beginning of time t we must have that $link_j \cap link_k = \emptyset$ since otherwise j could not have grown. But by the previous lemma, v is represented in both $link_j$ and $link_k$ at the beginning of time t by $\text{alias}(v, t-1)$. This contradiction establishes that L and R are disjoint. ∎

This theorem establishes the promised upper bound on the length of the *link* lists:

Corollary 2.1: bound $\leq \lceil \log n \rceil + 1$. ∎

This completes the proof that the filtration is correct and runs in time $O^*(n)$. Of course any reasonable sequential algorithm cm be used to find the connected components of the filtered output since it comprises no more than $O(n \log n)$ edges. A more attractive possibility is to use a postprocessing pass of the filtration algorithm in which the identity matrix is the input. This scheme sequentially causes each processor to broadcast (and thereby reduce to one entry) its *link* array. After the last row of the identity matrix has been processed, each list $link_j$ contains only one entry, and it is easily seen that for all j, the entry $link_j(0)$ is equal to cc(j). (We suggest to the reader the following question. After row i of the identity matrix is processed, *count;* = **1.** Is $link_i(0) = cc(i)$? We give a hint: if at some time some processor j has a *link* entry containing any name other than j, then no other processor's list contains the name j.)

An even more attractive possibility is to interleave this second pass with the filtration process. Then no postprocessing is required; after the last big time step, the connected components problem is solved. It is easy to verify that the interleaving neither affects the correctness nor the $0^*(n)$ time bound.

We remark that an analogous, but simpler, algorithm can be designed where the most popular processor k broadcasts only one name instead of its entire list, and no deletions from memory occur. In this case, however, the filtering is less efficient; each local memory may have to hold $0(\sqrt{n})$ entries; thus $O^*(n^{3/2})$ area is required.

2.5 Conclusion

In this chapter WC have illustrated a progression of successively better parallel connected components algorithms. The key elements in all of these algorithms are the ideas of filtration, data distribution, control of communication and the appropriate USC of mild redundance.

The unfortunate thing about these algorithms is that they give little guidance for constructing efficient algorithms for other problems. Our reliance upon deceitful filtration is particularly daunting; while deceit is fine for connected components, it is far from clear that it can be employed in more difficult graph problems. In the next chapter, we develop a paradigm, based on faithful filtration, that is readily adaptable to a much wider class of problems.

3. Funnelled-Pipeline Algorithms

3.1 Introduction

In this chapter we explore what we call the funnelled-pipeline paradigm. Circuits constructed with this paradigm make use of cascaded filtration. Thus such circuits are composed of a series of pipelined processors. Because each of these stages acts as a filter, the data flow decreases along the pipeline. The decreasing data flow is essential in that it allows successive filtration stages to run longer, and hence more thoroughly. This feature is a marked departure from conventional pipelines; transition times along our pipeline form an exponentially-increasing sequence.

We begin by examining in greater detail the idea of filtration. Then we present a general outline of the data flow and hardware architecture of the funnelled-pipeline model; finally we apply the paradigm to several specific graph problems.

3.2 Filtration

In the preceding chapter we exploited the idea of filtration. It is useful to define this idea in greater detail and formality.

For the sake of exposition, assume that we are given a fixed set, *V*, of vertices. Let S denote the powcrset of the set of all edges $\{u,v\}$ for u and v members of *V*. Thus S can be viewed as the family of all graphs on *V*. By a graph problem, we will mean a function P with domain S. The range of P depends on the problem. For instance, for the connected components problem, P maps members of S to functions cc : $V \rightarrow V$. In the minimum spanning forest problem, *P* maps S to S. Perhaps WC should note that some common problems are best described as relations rather than as functions. An example is the problem of finding a spanning forest of a graph; there are often several. However, in order to keep the notation simple, we restrict ourselves to functions; it is a simple matter to adapt what follows to relations.

We would like to define the properties that make something a filter. In keeping with our earlier use of the term, a filter is a map F : S → S. (Though, of course, several generalizations arc possible.) The obvious property that F should possess is that it should leave invariant the graph problem. That is, we require that $P(F(E)) = P(E)$ for all $E \in$ S.

That, however, is not quite enough for our purposes. Recall that our aim is to produce linear sized circuits for solving quadratic sized problems. This implies that our filters will at any time be operating only on a subgraph of the input graph. Therefore a filter must summarize each portion of a graph in a manner that permits solving the problem no matter what the rest of the graph may be. Thercforc WC propose the following definition.

Definition 3.1: $F : S \to S$ is a *filter* for P if $P(F(E) \cup E') = P(E \cup E')$ for all E and E' members of S.

Note that this definition implies that $P(F(E)) = P(E)$. Incidently, we can make precise the terms *faithful* and *deceitful*. A filter F is faithful provided that $F(E) \subseteq E$ for all $E \in S$. A filter that fails to be faithful is said to be deceitful. Since the point of filtration is to reduce the volume of data, we will generally be interested in filters F with the property that $|F(E)| \leq |E|$.

It may be worthwhile to consider a simple example. Let P be the connected components problem. Let F be defined so that for all $E \in S$ we have that $F(E)$ is a spanning forest of E. Then F is a filter for P. We leave it to the reader to verify this fact.

Because our algorithms will make use of cascaded filtration, we examine some further consequences of our dclinition. Suppose that F is a filter for P. We define a binary operation $B : S \times S \to S$ by $B(E_1, E_2) = F(E_1 \cup E_2)$. It will turn out that B captures the essence of the fundamental step of our algorithms, namely the operation of combining two subgrnphs and filtering the result.

By a cascaded filtration, we mean any repeated composition of the function B. We give the following inductive definition.

Definition 3.2: $\hat{B} : S^k \to S$ is a *cascaded filtration* provided that

$$\hat{B}(E_1, \ldots, E_k) = \begin{cases} F(E_1), & \text{if } k = 1; \\ B(E_1, E_2), & \text{if } k = 2; \\ B\left(\hat{B}_1(E_1, \ldots, E_m), \hat{B}_2(E_{m+1}, \ldots, E_k)\right), & \text{for some } m \text{ if } k > 2 \text{ and } \hat{B}_1 \text{ and } \hat{B}_2 \text{ are cascaded filtrations.} \end{cases}$$

The important property of a cascaded liltration is that it acts as a filter.

Theorem 3.1. **Cascaded Filtration:** Let $\hat{B}(E_1, \ldots, E_k)$ bc a cascaded liltration. Then $P(\hat{B}(E_1, \ldots, E_k) \cup E') = P(\cup_{1 \leq i \leq k} E_i \cup E')$ **for all** $E_1, \ldots, E_k, E'$ **in S.**

Proof 3.1: This theorem is proved by induction. It is vacuous for the case $k = 1$. **For** $k = 2$, we have $\hat{B}(E_1, E_2) = F(E_1 \cup E_2)$. Let $E = E_1 \cup E_2$. We must show that $P(\hat{B}(E_1, E_2) \cup E') = P(E \cup E')$. But $P(\hat{B}(E_1, E_2) \cup E') = P(F(E) \cup E') = P(E \cup E')$.

If $k > 2$ then $\hat{B}(E_1, \ldots, E_k) = B(\hat{B}_1(E_1, \ldots, E_m), \hat{B}_2(E_{m+1}, \ldots, E_k))$ **for some m.** Let $E = \cup_{1 \leq i \leq k} E_i$, $L = \cup_{1 \leq i \leq m} E_i$ **and** $R = \cup_{m < i \leq k} E_i$. We have

$$\begin{aligned} P(\hat{B}(E_1, \ldots, E_k) \cup E') &= P(B(\hat{B}_1(E_1, \ldots, E_m), \hat{B}_2(E_{m+1}, \ldots, E_k)) \cup E') \\ &= P(F(\hat{B}_1(E_1, \ldots, E_m) \cup \hat{B}_2(E_{m+1}, \ldots, E_k)) \cup E') \\ &= P(\hat{B}_1(E_1, \ldots, E_m) \cup \hat{B}_2(E_{m+1}, \ldots, E_k) \cup E') \\ &= P(L \cup R \cup E') \qquad \text{(induction on } \hat{B}_1, \hat{B}_2) \\ &= P(E \cup E'). \end{aligned}$$

∎

Cascaded filtration can be viewed as a general method for organizing parallel divide-and-conquer computations. Imagine that we wish to compute *P(E)* given some input graph *E*. Let each set E_i (for $0 \leq i < n$) consist of the edges in *E* joining vertex i to some vertex j where j > i. Then the Cascaded Filtration theorem guarantees that we can compute *P(E)* by computing $P(\hat{B}(E_0, \ldots, E_{n-1}))$. This latter computation of course could be accomplished by the obvious balanced binary tree of filter units. In the next section we describe a much more attractive implementation, the funnelled pipeline, in which each level of the binary tree is replaced by a single filter unit.

3.3 Funnelled-Pipeline Algorithm Structure

We assume, as before, that the graph problem is presented as an n-by-n adjacency (or edge-weight) matrix *A* and that, for simplicity, n is a power of two. The algorithm structure is composed of log n pipelined filter units. Each filter unit implements the filtration operation $B(E_1, E_2) = F(E_1 \cup E_2)$ where *F* is a filter for the graph problem. Thus, at each activation, each-Jilter unit receives as input the edge sets E_1 and E_2. It produces as output the edge set $B(E_1, E_2)$. This output set serves as one of the input sets in the next activation of the next filter unit along the pipe. Thus the first filter unit, filter zero, reads in a stream of n edge sets, namely the sets corresponding to the n rows of the adjacency matrix. It outputs a stream of $\frac{n}{2}$ "filtered" edge sets. With a slight abuse of notation, we have as its i^{th} output set, the set $B(A_{2i}, A_{2i+1})$. These $\frac{n}{2}$ sets are fed to the second filter unit which computes the $\frac{n}{4}$ sets $B(B(A_{4i}, A_{4i+1}), B(A_{4i+2}, A_{4i+3}))$. These $\frac{n}{4}$ sets of "doubly filtered" edges arc fed to the third filter unit, etc. Note that each output set of each filter unit is a cascaded filtration. In particular, the output of the last filter unit is a cascaded filtration of the entire input graph.

The first question that arises is that of representing edge sets. We enforce a strict convention. Each edge set will be represented by an n-element array $c_0, c_1, \ldots, c_{n-1}$. Each element c_i is itself a set of edges; however its size is bounded by a constant (in the following algorithms, that constant is one or two). Furthermore, we insist that if the edge {u, v} is a member of c_i, then $u = i$ or $v = i$. In other words, each edge must be stored at one of its endpoints; thus c is actually an adjacency-list rcprcsentation. Note that it is trivial to convert adjacency-matrix rows to this form. Note also that each edge set contains $O(n)$ edges. It should be admitted that this representation scvcrcly limits the types of filters that can bc employed. The advantage of the representation lies in its suitability for parallel architectures.

WC will require all filters to be faithful. This implies that the cascaded filtrations are also faithful. We describe an important consequence. Note that in the adjacency matrix, all edges in the set corresponding to row i are incident to vertex i. In other words, all edges on a given matrix row have a common endpoint, which we call the leader. Consider an edge set output by the first filter unit. Each edge in that set must be incident to at least one of the leaders of the two matrix rows from which it was derived. Thus each edge set output by the first unit has two leaders, namely the single leaders of the two corresponding

input rows. Similarly, the sets output by the second filter unit have four leaders, etc. The fact that each edge is incident to at least one leader will make rapid filtration possible.

WC list below some of the critical properties and terminology of the funnelled-pipeline structure. Note that vertex indices, filter indices, and activation counts are numbered beginning with zero. Recall that for convenience we have assumed that n is a power of two.

- Filter unit k is activated $n/2^{k+1}$ times, and thus produces $n/2^{k+1}$ output sets.
- Filter k is activated with a period of 2^{k+1} cycles. (A cycle is defined by the reading of a single row of the adjacency matrix into the input buffer of filter zero.)
- At activation i, filter k reads two edge sets. Each of these two sets contains 2^k leader vertices. The first set has as its leaders vertices $i2^{k+1}$ through $i2^{k+1} + 2^k - 1$. The second set has as leaders vertices $i2^{k+1} + 2^k$ through $(i + 1)2^{k+1} - 1$. Thus at activation i, filter k contends with a total of 2^{k+1} leaders, and each edge in the union of the two input sets is incident to at least one of them.
- During activation i, filter k produces one output set. This set contains 2^{k+1} leaders, namely vertices $i2^{k+1}$ through $(i + 1)2^{k+1} - 1$.
- At activation i of filter k, the vertices with index $(i + 1)2^{k+1}$ through n -- 1 are called *gangmembers*.
- During activation i of filter k, the vertices 0 through $i2^{k+1} - 1$ are called dead vertices. No edges among the input and output sets are incident to dead vertices.
- The last filter, filter $\log n - 1$, is activated once. Its output is a cascaded filtration of the entire input graph.

Each filter unit will be built as a linear array of n processors, one processor corresponding to each graph vertex. These processors will be interconnected by an ethernet. Thus each filter will have width O(n); as will the data streams connecting successive filters. See Figure 3.1. This architecture permits transferring the edge set output by one filter stage to the input buffer of the next in O(1) time steps. We will assume that each filter unit contains a simple input buffer capable of holding two edge sets.

3.4 A Minimum Spanning Forest Algorithm

This section describes a funnelled-pipeline algorithm for computing minimum spanning forests on undirected weighted graphs. An n vertex graph is represented by an n-by-n upper-triangular edge-weight matrix A. Entry A_{ij} represents the weight of the edge joining vertices i and j. Edge weights may, of course, be infinite, but must be specified by at most O*(I) bits.

It is convenient to have a notion of the *unique* minimum spanning forest. Thus edges with equal weights will be strictly ordered according to, say, the lexicographic ordering of their endpoints. It can be seen (in a variety of ways) that such a scheme induces a strict ordering on the total weights of spanning forests.

Our first task is to find a suitable filter. For any set of edges E, let $MSF(E)$ denote their minimum weight spanning forest. We present two facts.

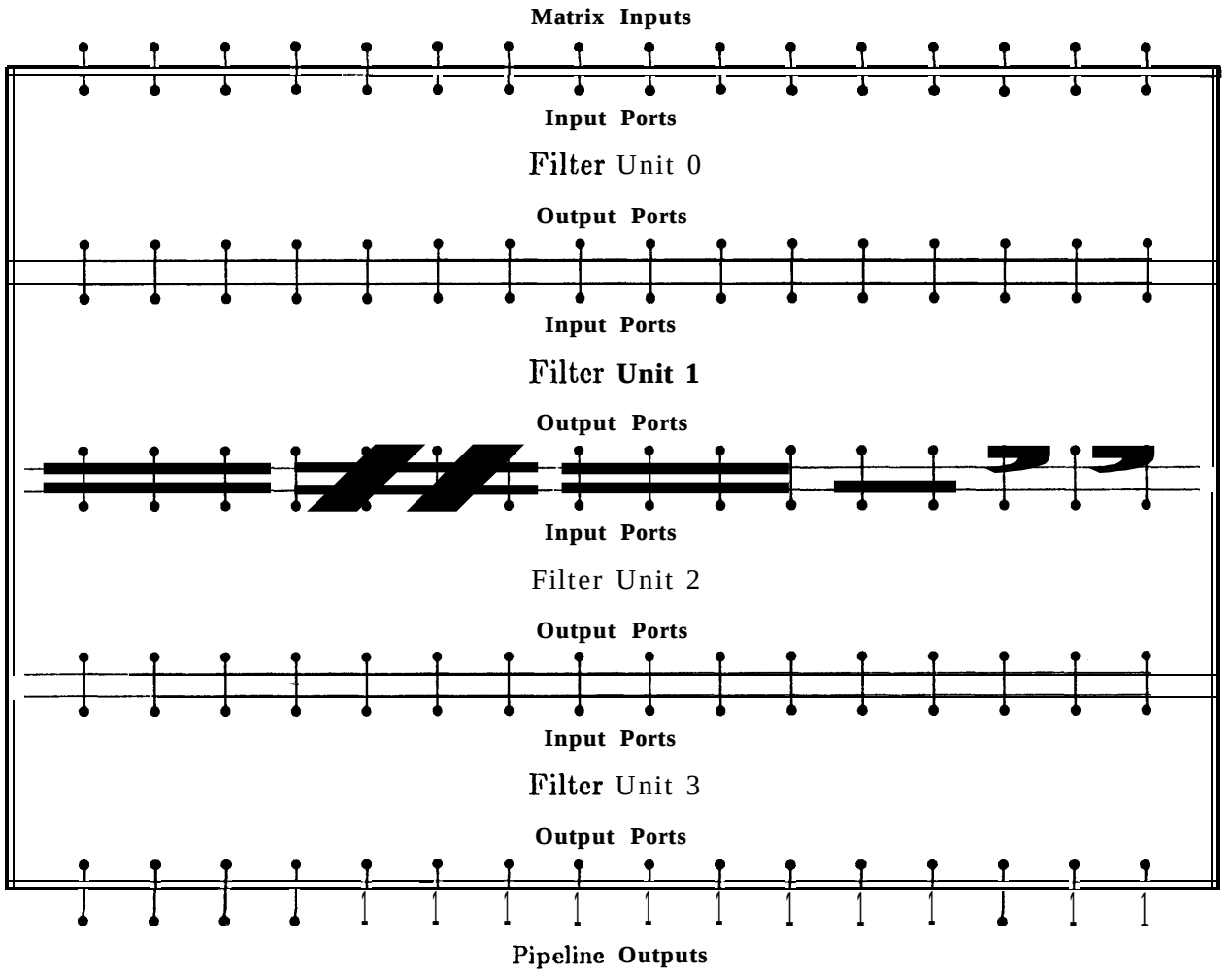

Figure 3.1: Funnelled Pipeline

Lemma 3.1: Let $e \in E$. Then e $\in$ MSF(E) if and only if e is not the edge of maximum weight in any cycle of E containing e.

Lemma 3.2: If e = $\{u, v\} \in E$ but e $\notin$ MSF(E) then there is a path in MSF(E) joining u to v, and each of that path's edges has weight less than the weight of e.

We obtain the following corollary.

Corollary 3.1: The function MSF is itself a faithful filter for the minimum spanning forest problem.

Proof 3.2: We must establish that MSF(MSF(E) $\cup$ E') = MSF$(E \cup E')$. Let e be any edge in $E \cup E'$. Suppose that e $\notin$ MSF (MSF (E) $\cup$ Id). Then Lemma 3.2 implies that there exists a path in MSF(E) $\cup$ E' joining the endpoints of e with the property that all of the path's edges have weight less than the weight of e. That path must of course also lie in $E \cup E'$, and thus Lemma 3.1 implies that e $\notin$ MSF$(E \cup E')$.

On the other hand, suppose that e $\notin$ MSF(E ∪ E). Evidently there is a cycle e, e_1, e_2, ..., e_k in $E \cup E'$ of which e has maximum weight. For each edge $e_i = \{u, v\} \in E$ we can select a path in MSF(E) joining u to v wherein each edge has weight at most that of e_i. Hence e has maximum weight in a cycle in MSF(E) ∪ E'. Lemma 3.1 then implies that e $\notin$ MSF(MSF(E) ∪ E'). ∎

One further fact about minimum spanning forests will be required to verify our algorithm.

Lemma 3.3: Let (V, E) be a graph. Let H be a subset of V and let C = {e $\in E$ | e joins a vertex in H to a vertex in $V - H$}. Let f be the edge of least weight in C. Then f is a member of MSF(E).

We note that our filter has an important property. Its output is a forest, and hence contains at most $n - 1$ edges. Furthermore, each tree can be rooted and each edge can be stored at its child vertex endpoint. This permits adherence to our convention for representing and transmitting edge sets among the filter units. In each edge set, there will be stored at most one edge per vertex.

Each filter in our pipeline implements an algorithm that computes the minimum spanning forest of the union of two input edge sets. In fact, the same algorithm is employed in each of the log n filter units. Each activation of filter unit k must take at most $O(2^{k+1})$ steps. In this regard, the fact that each activation contends with 2^{k+1} leaders will prove critical.

The computation for activation i of filter unit k proceeds as follows. There arc three major steps. The first is to find, for each gangmember, the minimum weight incident edge (if any). Such edges arc guaranteed by Lemma 3.3 to lie in the minimum spanning forest of the input edge sets. Because every input edge is incident to at least one leader, each of the selected minimum weight edges connects a gangmember to a leader. Thus this first step can bc viewed as the formation of gangs of gangmember vertices, each gang being led by a leader vertex. This gang-formation step must be done with a high degree of parallelism, since, in general, the number of gangmembers vastly exceeds the time budget of $O(2^{k+1})$. Observe that after these edges have been selected, most of the minimum spanning edges have been found. If $l = 2^{k+1}$ is the number of leaders, the minimum spanning forest consists of at most l -- 1 additional edges.

Thus the second step chooses at most $l - 1$ more edges to link together the gangs. There is sufficient time to select these edges sequentially. Indeed, the algorithm for this step differs little from the conventional "greedy" algorithm.

The final step consists of rearranging the now-complete minimum spanning forest so as to obey our convention for representing edge sets. This is accomplished by traversing the forest and storing each edge at its child endpoint. Again a high degree of parallelism is required. It is obtained by using a graph traversal that never visits leaf vertices that are gangmembers. Since there are 1 leader vertices, any tree, and hence any forest, has at most 21 internal vertices. This is due to the fact that the children of every internal gangmember vertex must bc leaders.

We now present a more detailed description of the algorithm. Each processor i will contain a list E_i of at most two edges. These edges will be stored in records containing the following fields.

- this-end: One endpoint of the edge.
- other-end: Other endpoint of the edge.
- *weight:* Weight of edge.
- *edge-status:* Current status of this edge.
- *other-status:* Status of *other-end* vertex.
- *other-leader:* Leader associated with vertex *other-end.*

The *field this-end* is a notational convenience; if $e \in E_i$ then $e.this\_end = i$. Each processor i will contain registers $status_i$ and $leader_i$, as well as a few temporary locations.

The main program is as follows.

```
program filterunit (k, n);
begin
  co k is the filter number, n is the graph size oc;
  for i:=0 to n/2^(k+1) do
    co i is the activation number oc;
    call initialize;
    call join-gang;
    call merge-gangs;
    call rearrange;
    call output
    od
end jilterunit.
```

The procedure *initialize* reads the two input edge sets and initializes status registers. The *edge-status* field of each edge is set to *standby;* initially each edge is a potential member of the minimum spanning forest. The status of each vertex is set to *ungrabbed;* this value identifies vertices that have not yet been visited in the traversal of the minimum spanning forest.

```
procedure initialize;
begin
  for all v, 0 ≤ v < n do in parallel
    status_v := ungrabbed;
    leader_v := v;
    read edges from input buffer into E_v;
    for all e ∈ E_v do
      co note that there are at most two edges in E_v oc;
      e.other_status := ungrabbed;
      e.edge_status := standby;
      e.other_leader := e.other-end
      od
    od
end initialize;
```

The procedure *join-gang* is responsible for linking each gangmember to a leader via the edge of least weight. Note that an edge joining gangmember g to leader l is stored either at g or at 1. Hence, we first broadcast all edges stored at the leaders in order that the gangmembers bc able to pick their cheapest incident edge. Then we again sequentially examine the edges stored at the leaders so as to appropriately adjust their status fields; whenever an edge is determined to lie in the minimum spanning forest the corresponding *edge-status* field is set to *selected*. The entire *join-gang* procedure takes at most $O(2^{k+1})$ steps since there are at most two edges stored at each of the 2^{k+1} leaders.

```
procedure join-gang;
begin
  co find the cheapest edge stored locally at gangmembers oc;
  for all gangmembers g do in parallel
    if g has any stored edges then
      let cheapest_g be such that cheapest_g.weight = min{f.weight | f ∈ E_g} else
      cheapest_g := nil fi
    od;
  co find cheapest edge not locally stored oc;
  for all leaders l do
    for all e ∈ E_l do
      if cheapest_{e.other_end}.weight > e.weight then cheapest_{e.other_end} := e fi
      od
    od;
  co we now have found cheapest edge incident to each gangmember oc;
  co update status of all locally stored cheapest edges oc;
  for all gangmembers g do in parallel
    if cheapest_g ∈ E_g then
      cheapest_g.edge_status := selected;
      leader_g := cheapest_g.other_end fi
    od;
  co update status of all non-locally stored cheapest edges oc;
  for all leaders l do
    for all e ∈ E_l do
      if there exists a gangmember g such that cheapest_g = e then
        e.edge-status := selected;
        leader_{e.other_end} := e. this-end fi
      od
    od;
  co update other-leader fields for all edges stored at leaders oc;
  for all leaders l do
    for all e ∈ E_l do
      e.other_leader := leader_{e.other_end}
      od
    od
end join-gang;
```

Procedure *merge-gangs* links together the gangs by a greedy algorithm. It starts at an arbitrary gang. It then selects the cheapest "candidate" edge joining this gang to another. The new gang is merged into the first gang and the process is repeated for this new super-gang. Merging continues until the super-gang is as large as possible. This entire process is repeated for each connected component of the graph. The gangs that have not yet been merged are identified by leader vertices v for which $status_v = ungrabbed$. Edges that join *ungrabbed* vertices to *grubbed* vertices are identified by the value *candidate* in their *edge-status* field. Note that at most a total of $2^{k+1} - 1$ gang merges are required to complete the minimum spanning forest.

```
procedure merge-gangs;
  procedure visit (i, newleader);
  begin
    co incorporate the gang containing i into the gang of newleader oc;
    oldleader := leader_i;
    co grab everybody who is in oldleader gang oc;
    for all v, 0 ≤ v < n do in parallel
      if leader_v = oldleader then
        status_v := grabbed;
        leader_v := newleader
        fi
      od;
    co update other-end status fields in edge records oc;
    for all v, 0 ≤ v < n do in parallel
      for all e ∈ E_v do
        if e.other_leader = oldleader then
          e.other-leader := newleader ;
          e.other_status := grabbed
          A
        od
      od;
    co update edge-status fields oc;
    for all v, 0 ≤ v < n do in parallel
      for all e ∈ E_v do
        case
          e.edge_status = standby ⟹
            if status_v ≠ e.other_status then e.edge_status := candidate fi;
          e.edge_status = candidate ⟹
            if status_v = e.other-stutus then e.edge_status := useless fi
          endcase
        od
      od
  end visit;
```

```
function nestedge;
begin
  co select cheapest candidate edge oc;
  if there exist any v and e ∈ E_v such that e.edge-status = candidate then
    let f be such that j ∈ {e ∈ E_v | 0 ≤ v < n} and f.weight = min{e.weight | e ∈ E_v and 0 ≤
    v < n};
    return f
    else return nil fi
end nextedge;
begin merge-gangs
  while there exists a leader r with status_r = ungrabbed do
    co start a tree rooted at r oc;
    call visit(r, r);
    while nextedge ≠ nil do
      co merge another gang into tree rooted at r oc;
      f := nextedge;
      f.edge_status := selected;
      if f.other_status = grabbed then visit(f.this-end,r) else visit(f.other_end, r) fi
      od
    od
end merge-gangs;
```

Procedure rearrange rearranges the selected minimum spanning forest edges so as to store at most one at each vertex. This is accomplished by first computing the degree of each gangmember in the minimum spanning forest. This identifies gangmember leaves. Then all internal vertices (and leaf leader vertices) are traversed, and all edges are moved (if necessary) to their child cutpoints. This procedure takes $O(2^{k+1})$ steps.

```
procedure rearrange;
  procedure mark-leaves;
    co decide which vertices are leaves and which are internal nodes of the forest oc;
    co compute the degree of each gangmember vertex oc;
    for all gangmembers g do in parallel
      degree_g := |{e ∈ E_g | e.edge-status = selected}|
      od;
    for all leaders l do
      for all e ∈ E_l do
        if e.edge_status = selected then degree_{e.other_end} := degree_{e.other_end} + 1 fi
        od
      od;
    co set vertex type fields oc;
    for all gangmembers g do in parallel
      if degree_g ≤ 1 then type_g := don't-visit else type_g := do-visit fi od;
    for all leaders l do in parallel type_l := do-visit ad;
  end mark-leaves;
```

```
  procedure traverse(i);
  begin
    co visit vertex i oc;
    status_i := finished;
    co deal with vertices storing an edge to i oc;
    for all v, 0 ≤ v < n do in parallel
      if there exists e ∈ E_v such that e.other_end = i and e.edge_status = selected then
        out_v := e;
        e.edge-status := unselected;
        if type, = don't-visit then status,, := finished else status_v := frontier fi
        fi
      od;
    co deal with any selected edges stored at i oc;
    for all e ∈ E_i do
      if e.edge-status = selected then
        m := e.other_end;
        out, := e;
        e.edge_status := unselected;
        if type, = don't-visit then status,,, := finished else status,,, := frontier fi
        A
      od
  end traverse;

begin rearrange
  co mark the leaves oc;
  call mark-leaves;
  co set vertex status fields oc;
  for all v, 0 ≤ v < n do in parallel' status_v := unvisited od;
  co traverse the minimum spanning forest oc;
  while there exists a leader r with status, = unvisited do
    co traverse the tree rooted at r oc;
    call traverse(r);
    while there exists a vertex v such that status_v := frontier do traverse(v) od
    od
end rearrange;
```

Procedure output simply writes the rearranged minimum spanning forest edges to the output buffer.

```
procedure output;
begin
  for all v, 0 ≤ v < n do in parallel write out,, to output port, od
end output;
```

A careful examination of the above algorithm reveals that each activation of filter unit k requires time $0^*(2^k)$ on an ethernet architecture. Since filter unit k is activated every $n/2^k$ cycles, the stages of the pipeline are properly balanced, and we have a when- and where-determinate minimum spanning forest algorithm requiring O*(n) time and area. Observe that no postprocessing is required; the output of the last filter unit is the minimum spanning forest of the input graph.

3.5 A Funnelled-Pipeline Algorithm for the Biconnected Components Problem

In this section we describe a funnellcd-pipeline algorithm that solves the biconnected components problem. Let G = (V, E) be a graph. Define a relation $\star$ on the edges so that $e_1 \star e_2$ if $e_1 = e_2$ or if there is a simple cycle (*i.e.* a cycle without repeated vertices) in G containing e_1 and e_2. It can be shown that $\star$ is an equivalence relation. Let $E_1, \ldots, E_k$ be the equivalence classes of E under $\star$. Let V_i be the set of endpoints of edges in E_i. Then the sets V_i are the *biconnected components* or blocks of G. We refer the reader to [AHU] for further background discussion.

Definition 3.3: Suppose that E is a graph. We say that a graph E' *represents* E **if** the following three conditions hold.

- E' is a subgraph of E.
- The connected components determined by E' are identical to those dcternrined by E.
- Every pair of vertices x and y lie in a simple cycle of E' if (and only if) they lie in **a** simple cycle of E.

Theorem 3.2: If a subgraph E' represents a graph E, then E' and E possess identical biconnected components.

Proof 3.3: It suffices to show the one non-trivial direction. Suppose that B is the set of vertices of a biconnectcd component in E, and x and y are members of B. Then **there is** a simple cycle in E, and hence a simple cycle in E', containing x and y. Thus x and y lie in one uniquely determined biconnected component of E' (two biconnectcd components intersect in at most one vertex). Hence all members of B lie in that biconnected component of E'. ∎

We next demonstrate that depth-first search can be used to find representing graphs.

Definition *3.4:* Let E be a graph. Let *T(E)* contain the edges of a depth-Grst traversal of E. That is, the forest $T(E)$ consists of the subset of the edges in E traversed by some depth-first search of E. Furthermore, for every vertex v, let $H(v)$ bc the highest incident back edge of E (if any) induced by the traversal $T(E)$. (The highest back edge from x is the one to the highest ancestor of x (in the forest *T(E))* reachable by a back edge (in *E)* from x.) Let $F(E) = \bigcup_v H_v \cup T(E)$.

Theorem 3.3: The graph *F(E)* represents E.

Proof *3.4:* Note first that $F(E)$ is a subgraph of E. $T(E)$ and E possess identical connected components, as do $T(E)$ and $F(E)$; hence $F(E)$ and E possess identical connected components.

Suppose that vertices x and y lie in a simple cycle of E. Assume without loss of generality that x is an ancestor of y in the depth-first traversal $T(E)$. Note that x and y lie in the same biconnected component of E. Therefore no vertex on the path from x to y in the depth-first traversal is an articulation point. Thus there is a directed path from y to x consisting only of highest back edges (oricntctl from descendent to ancestor)

and depth-first traversal edges (oriented from ancestor to descendent). A subset of these edges combined with an appropriate subset of the remaining depth-first traversal edges yield a simple cycle containing x and y. ∎

We now show that filtration can be accomplished by finding representations. We make use of the following lemma.

Lemma 3.4: Let E be a graph. Suppose that $k > 2$ and that $v_0, \ldots, v_{k-1}$ is a sequence of distinct vertices with the property that for all i either $\{v_i, v_{i+1 \bmod k}\} \in E$ or v_i and $v_{i+1 \bmod k}$ both lie in the same biconnected component of E. Then all v_i lie in a single biconnected component of E.

Proof 3.5: If the v_i fail to lie in a single biconnected component, there would exist in E a simple cycle of edges passing through more than one biconnected component. This would contradict the definition of biconnected components. ∎

Theorem 3.4: Suppose that R represents E. Then $R \cup E'$ represents $E \cup E'$ for all sets of edges E'.

Proof 3.6: It suffices to show that whenever two vertices x and y lie in a simple cycle C of $E \cup E'$, then they also lie in a simple cycle of $\hat{R} = RUE'$. So consider an edge e = {u, v} that lies in C but not in $\hat{R}$. Thus e is in E but is not in R. Since R represents E and u and v are in the same connected component of E, there must be a simple path in R connecting them. Because R is a subgraph of E, this path must also be in E. Hence u and v lie in a simple cycle of E, and again, since R represents E, they lie in a simple cycle of R. Thus u and v arc in the vertex set of one biconnected component of $\hat{R}$; the cycle C satisfies the conditions of the previous lemma. Therefore the vertices of C must lie within a single biconnected component of $\hat{R}$. This ensures that x and y lie on a simple cycle of $\hat{R}$. ∎

This theorem establishes that the function F defined above is a filter for the biconnected components problem. We make use of F in constructing a funnelled-pipeline biconnected components algorithm. Note that the edge set $F(E)$ can be stored with at most two edges per vertex. At each vertex we store the depth-first edge from its parent and the back edge to its highest reachable ancestor. Thus back edges are stored at the tail vertex; tree edges at the child.

The algorithm is structured in the same way as the minimum spanning forest algorithm. It consists of a funnelled pipeline of log n filtering stages. Each stage repeatedly combines two subgraphs and filters the union. The resulting set of output edges arc passed on to the next stage.

Because of the similarity between the structure of this algorithm and that of the minimum spanning forest algorithm, we present only a description of the filtration stages, and leave it to the reader to fill in the rest.

At activation i, every stage k in the pipeline receives two input sets. Each set contains at most two edges per vertex. The filtration stage forms the union of the two graphs given by the input sets. It performs a depth-first traversal on this union and discards all edges

which are neither tree nor highest back edges. The remaining edges are shipped to the next stage.

The filtration depends upon performing a depth-first traversal of the union of the input graphs. Like the restructuring operation used in the minimum spanning forest algorithm, the depth-first traversal must avoid making unueccssary visits to leaf vertices. Thus it will not visit gangmember leaves. Such a traversal can be accomplished in time proportional to the sum of the number of internal vertices and the number of leaves that are leaders in the induced depth-first spanning forest. This quantity is bounded by the sum of the number of leaders and the number of internal gangmember vertices. Because each internal gangmember vertex has at least one child, and each such child must be a leader, there can be no more internal gangmember vertices than there are leaders. Hence at stage k no more than 2^{k+2} vertices are visited in the depth-first traversal.

In order to avoid visiting gangmember leaf vertices during the depth-first traversal, each gangmember g keeps a count *unvisited-neighbors!,* of its as yet unvisited neighbors. Initially *unvisited-neighbors,* is the total degree of vertex g in the input graph. Each time that a neighbor of g is visited during the traversal, the register $unvisited\_neighbors_g$ is decremented. If $unvisited\_neighbors_g$ reaches zero before g is visited, then g has become a leaf of the depth-first search tree. Otherwise, if g is visited when $unvisited\_neighbors_g > 0$, then g becomes an internal vertex.

Thus the filtering algorithm for every activation consists of the following steps.

(1) The leader vertices sequentially broadcast their names and adjacency lists. Each gangmember g counts the number of times it hears its own name broadcast. The degree of g, and hence the initial value of *unvisited-neighbors,,* is then g's computed count plus the number of edges stored at the processor corresponding to g.

(2) The depth-first search starts with some leader. When visiting a vertex y, the name y and y's adjacency list arc broadcast to the other processors. Any processor (i.e. vertex) x which is not yet visited, and which recognizes y in its locally-stored adjacency list, or hears its name in the broadcast list, makes note of y's name, aud, if x is a gangmember, decrements $unvisited\_neighbors_x$. An unvisited x need only retain the first and last names noted. The last name thus noted by a processor x before it is visited (if ever) is, of course, its parent in the depth-first search tree. This name is retained to specify the tree edge bctwccn x and its parent. The first name noted is also retained if different from x's parent; it is then the highest ancestor reachable by a back edge from x.

(3) The next vertex to visit in the depth-first search is selected arbitrarily from among the unvisited vertices v adjacent to the current vertex which are not yet gangmember leaves (i.e. for which *unvisited-neighbors,* $\neq 0$). If there arc no such vertices, the traversal backs up to the parent of the last visited vertex. If this last vertex was a root and there arc still unvisited leaders, the traversal of a new tree is started.

Since the time for the depth-first traversal is O*(Z) where l is the number of internal nodes, the time for every activation of stage k is $O^*(2^k)$. Hence the overall timing is the same as for the minimum spanning forest algorithm, as is, for that matter, the space.

3.6 Conclusion

Our algorithms demonstrate the power of filtration and funnelled pipelining as well as some of the hidden parallelism intrinsic to many of the classical graph problems. Within this framework, trade-offs are of course possible between area and time requirements. For example, the log n filter units used in the funnelled-pipeline algorithms could be combined into a single unit, thereby reducing area at the expense of a factor of O(log n) in time.

Much more radical departures also present themselves. We discuss this topic in Chapter 5.

The funnelled-pipeline techniques illustrated in this chapter can be applied to a variety of graph problems to obtain solutions superior to those previously published [L-V]. Such problems, for which we obtain when- and where-determinate, $A = T = O^*(n)$, one-pass ethernet implementations, include:

- Connected Components
- Spanning Tree
- 2- Colorability
- Has a cycle
- Has an Eulerian cycle
- Minimum Spanning Forest
- Biconnected Components
- Planarity Testing

The minimum spanning forest algorithm can be simplified to yield funnelled-pipeline algorithms for both the connected components problem and the spanning tree problem. Funnclled-pipeline algorithms for solving the 2-colorability, cycle detection and Eulerian cycle detection problems can be obtained by augmenting the spanning tree algorithm. Planarity testing serves as the motivating problem for the next chapter.

4. Planarity and Representation

4.1 Introduction

This chapter is motivated by the problem of testing whether a given graph is planar. We assume that the graph is presented as an n-by-n adjacency matrix. The matrix is to be read row by row (with a when- and where-determinate schedule) into a circuit of area O*(n) that will be allowed *O*(n)* time to determine whether the graph is planar.

It will be seen that solving this problem effkiently in the parallel environment of interest depends upon solving a more fundamental problem of data rearrangement. In this chapter we develop a solution to that problem. We also briefly discuss its relationship to several other classic-al graph- theoretic problems.

4.2 Sparse Graphs and Their Representation

We noted that the crux of the problem of testing graph planarity lies in converting an adjacency-matrix graph representation into something more economical. To this end, WC develop an algorithm that transforms the n^2 adjacency-matrix bits into a linear-sized adjncency-list representation. Once these adjacency lists are stored, any of a variety of 0* (n)-time sequential planarity-testing algorithms can be cmployed. Thus the problem of testing graph planarity in the parallel environments discussed here reduces to a problem of data rearrangement.

Definition 4.1: A graph G = (V, E) is called *c-sparse* provided that every vertex-induced subgraph G' = (V', E') of G satisfies the relation $|E'| \leq c|V'|$.

Note that every planar graph is 3-sparse (Euler's relation), and, for another example, every tree is 1-sparse.

Definition 4.2: A graph G = (V, E) has a *c-representation* if it is possible to place its edges in adjacency lists A_v for v $\in V$ subject to the following constraints:

- For each edge e = $\{u, v\}$ in E, either e $\in A_u$ or e $\in A_v$.
- For each v $\in V$ the relation $|A_v| \leq c$ holds.

Observe that our minimum spanning forest funncllcd-pipclinc algorithm made use of 1-representations for storing and transmitting edge sets among the filter stages. The biconnec ted components algorithm used 2-reprcscntations.

Theorem 4.1: Every c-sparse graph has a [cl-representation,

Proof 4.1: This lemma is verified by induction on the number of edges in the graph. Clearly a graph devoid of edges has a [cl-representation. So suppose that G = *(V, E)* is c-sparse and that the family A_v, for v $\in V$ is a [cl-representation for G. Suppose further that $e = \{u, v\} \notin E$ and that $G' = (V, E \bigcup \{e\})$ is also c-sparse. We show that G' has a $\lceil c \rceil$-representation. The argument is reminiscent of the augmenting-path techniques employed in matching problems; this apparent coincidence is discussed later.

We find it useful to introduce the following definition. Let S be a set of vertices in G. Then let *A(S)* contain the new vertices named in adjacency lists associated with vertices in S. That is,

$$A(S) = \{x \notin S \mid \{s, x\} \in A, \text{ for some } s \in S\}.$$

We wish to consider a breadth-first traversal starting at the set $\{u, v\}$. Thus we define the sequence of sets V_i as follows:

$$V_1 = \{u, v\}$$
$$V_{i+1} = V_i \cup A(V_i).$$

There are two cases to consider. First, suppose that there is a vertex w and a least integer $k > 0$ such that w $\in V_k$ and $|A_w| < \lceil c \rceil$. In this case there is a path from u or v to w. That is, there is a sequence of edges $e_1, \ldots, e_{k-1}$ and vertices $v_1 \in V_1, \ldots, v_k \in V_k$ such that

- $v_1 \in \{u, v\}$
- $v_k = w$
- Each edge e_i joins v_i to v_{i+1} and $e_i \in A_{v_i}$

Now by "reversing" this path, we can make room for the new edge e. Let

$$A'_{v_1} = A_{v_1} \cup \{e\} - \{e_1\}$$
$$A'_{v_i} = A_{v_i} \cup \{e_{i-1}\} - \{e_i\}, \qquad 1 < i < k$$
$$A'_{v_k} = A_{v_k} \cup \{e_{k-1}\}$$
$$A'_x = A_x, \qquad x \notin \{v_i \mid 1 \leq i \leq k\}.$$

Now $|A'_{v_i}| = |A_{v_i}| \leq \lceil c \rceil$ for $i < k$ and $|A'_{v_k}| = |A_{v_k}| + 1 \leq \lceil c \rceil$ hence the family A' is a $\lceil c \rceil$-representation for G'.

If the first case does not hold, let k bc the smallest integer such that $A(V_k) = \emptyset$. Now consider the subgraph $H = (V_k, \bigcup_{w \in V_k} A_w)$ of G. Evidently H contains $|V_k|$ vertices and $\lceil c \rceil |V_k|$ edges. Hence the subgraph $(V_k, (\bigcup_{w \in V_k} A_w) \bigcup \{e\})$ of G' contains one more edge and thus fails to bc c-sparse. This contradiction establishes the lemma. ∎

Our use of sparse representations for sparse graphs is motivated by the requirements of distributed computation. In order to make effective use of parallel environments it is necessary to distribute the data to be processed uniformly among the available processing clcments.

The essence of our parallel planarity algorithm is to compute rapidly a sparse representation of the input adjacency matrix. As noted before, once this representation has been obtained, even conventional sequential algorithms will suffice to determine planarity in 0* (n) time and area.

We have established that if a graph is c-sparse it has a [cl-representation (thus planar graphs have 3-representations). However, it is far from clear that there exist efficient parallel algorithms for computing $\lceil c \rceil$-representations of c-sparse graphs. Discussion of this topic will be deferred to the end of this section. We will now present instead algorithms for computing &-representations of c-sparse graphs. Observe that such representations are equally suitable for the intended application.

4.3 Finding 2*c*-representations of c-sparse Graphs

In this section we will assume that the input graph is c-sparse. The algorithms may fail if this condition is not satisfied; however it is easy to augment these algorithms so that they detect such failure.

The following sequential algorithm finds 2*c*-representations of c-sparse graphs.

input $G_0 = (V, E)$;
do until $V = \emptyset$
  Pick some $v_0 \in V$ such that the degree of v_0 is no more than $2c$;
  $A_{v_0} := \{\{w, v_0\} \in E \mid w \in V\}$;
  $E := E - A_{v_0}$;
  $v := v - \{v_0\}$;
  od;
end.

The key to the success of this algorithm is the fact that every c-sparse graph (and hence cvcry subgraph of a c-sparse graph) contains at least one vertex of degree at most $2c$.

In order to parallelize this approach, wc will employ a funnelled-pipeline construction, The input to stage k at each activation will consist of two arrays of adjacency lists. Each adjacency list will have at most $2c$ entries; thus each processor will be presented with a total of at most $4c$ edges. Recall that there are 2^{k+1} leaders and that cvcry edge is incident to at least one leader vcrtcx. Stage k most rearrange the edges so as to produce a single array of adjacency lists, each list of length at most $2c$. This computation must be performed in time $O^*(2^k)$.

The major difficulty in adapting the above sequential algorithm lies in the fact that there may be a large number of vertices with low (*i.e.* at most $2c$) degree. Thus it would not do to visit each sequentially. A less scvcrc difficulty arises from our desire to produce

an algorithm suitable for ethernet communication. This restricted model precludes such operations as computing the degree of leaders by means of an addition census function.

In order to overcome these difficulties, we partition the vertices and edges as follows. Let the $l = 2^{k+1}$ leaders form the set L. Partition the gangmembers M according to their degree:

$$M_S = \{m \in M \mid degree(m) \leq 2c\}$$
$$M_B = \{m \in M \mid \text{degree(m)} > 2c\}.$$

Let E be the set of input edges. Partition E as follows.

$$E_S = \{e \in E \mid e \cap M_S \neq \emptyset\}$$
$$E_B = \{e \in E \mid e \cap M_B \neq \emptyset\}$$
$$E_L = \{e \in E \mid e \subset L\}.$$

We would like to bound the number of high-degree gangmembers. Consider the bipartite subgraph $(L \bigcup M_B, E_B)$. By sparseness, $|E_B| \leq c|L \bigcup M_B|$. However, by the definition of M_B, the relation $|E_B| > 2c|M_B|$ holds. Hence $2c|M_B| < c(l + |M_B|)$ and thus $|M_B| < l$.

This fact suggests a two phase algorithm. In the first phase, vertices in M_S *axe* identified and their edges (*i.e.* those in E_S) are disposed of (by being stored in the adjacency lists associated with the vertices in *MS).* The second phase is responsible for allocating the remaining edges among the remaining vertices. In this phase there are at most $2l$ vertices, namely $L \bigcup M_B$, and hence at most $2cl$ edges. An outline of the algorithm for stage k of the pipe follows.

(1) Compute. the degree of each gangmember

- This is accomplished by broadcasting all edges stored at leaders. Each gangmember counts the number of such edges incident to itself and adds this count to the number of edges stored locally.
- There are at. most $4cl$ edges to broadcast, hence this step requires $O^*(cl)$ time.

(2) Deal with the vertices in M_S and the cdgcs in E_S

- Again broadcast all edges stored at leaders. Each member of M_S (*i.e.* gangmembers with degree at most $2c$) stores each broadcast incident edge locally and sends a message to the broadcasting leader vertex. That leader deletes the edge from its local memory. Thus this stage transfers all edges in E_S to the processor corresponding to their endpoint in M_S.
- This step requires $O^*(cl)$ time.

(3) Deal with the vertices in $M_B \bigcup L$ and the edges in $E_B \bigcup E_L$

- There arc at most $2l$ vertices and $2cl$ edges left
- The following algorithm can bc employed to distribute these edges appropriately:

```
Make all vertices in M_B ⋃ L awake;
Compute the degree of each vertex by broadcasting all remaining edges;
do until all vertices are asleep
  select an awake vertex v with degree(v) ≤ 2c;
  move all cdgcs incident to v to v's local memory;
```

put v to sleep;
recompute degree of all awake vertices by broadcasting all edges removed (*i.e.* all edges now stored at v);
od;

- This step requires $O^*(cl)$ time.

Thus WC have an $0"(c2^{k+1})$-time algorithm for the k^{th} stage of the pipeline. Each of the n vertex processors needs O(c) storage; thus each stage requires O*(n) area. (Note that all required communications can be handled by an ethernet or bus linking the n processors.) Hence there is an $A = T = O^*(cn)$ when- and where-determinate algorithm for computing S-representations of c-sparse graphs.

This establishes the following theorem.

Theorem 4.2: There exists an $A = T = 0^*$ (cn) when- and where-determinate funnelled-pipeline algorithm for testing graph planarity.

4.4 The Complexity of Finding Representations

In this section we discuss the complexity of finding representations of sparse graphs. We find that both the parallel and sequential complexities of this problem are closely related to those of the well-known combinatorial problems of finding maximum network flows and matchings. This relationship suggest that developing a good parallel representation algorithm is at least as difficult as developing a good parallel bipartite matching algorithm. Although we are primarily interested in algorithms requiring moderate area and moderate time, we point out that the preceding observation applies also to extremely fast (poly-log time) algorithms.

WC first demonstrate that the c-rcprcscntation problem can be reduced to a network flow problem. It will bc convenient to restrict c to the domain of the integers. Define a transformation from the graph $G = (V, E)$ to the graph $G' = (V', E')$ as follows. Let V' contain a vertex $\hat{v}$ corresponding to each vertex $v \in V$, a source vertex s, a sink t, and a vcrtcx $\hat{e}$ corresponding to each $e \in E$. Thus $|V'| = |V| + |E| + 2$. The set E' contains three classes of edges. Each vcrtcx in V' corresponding to a vertex in V is joined by an edge of capacity c to the sink t. Each vertex in V' corresponding to an edge in E is connected by an edge of capacity one to the source s. Finally each vertex $\hat{e}$ in V' corresponding to an edge $\{u, v\}$ in E is connected by edges of unit capacity to vertices $\hat{u}$ and $\hat{v}$. Thus $|E'| = 3|E| + |V|$. An example of the transformation is shown in Figure 4.1.

Theorem 4.3: Let f be an integer valued maximum flow function for G'. Then the total flow from s to t under f is $|E|$ if and only if G has a c-rcprcscntation. Furthermore such maximum flows f arc in one-to-one correspondence with the c-representations of G.

Proof 4.2: Observe that since all edge capacities are integer, there is an integer valued maximum flow. Also note that since the total capacity of the edges incident to s is $|E|$, the maximum total flow cannot exceed $|E|$.

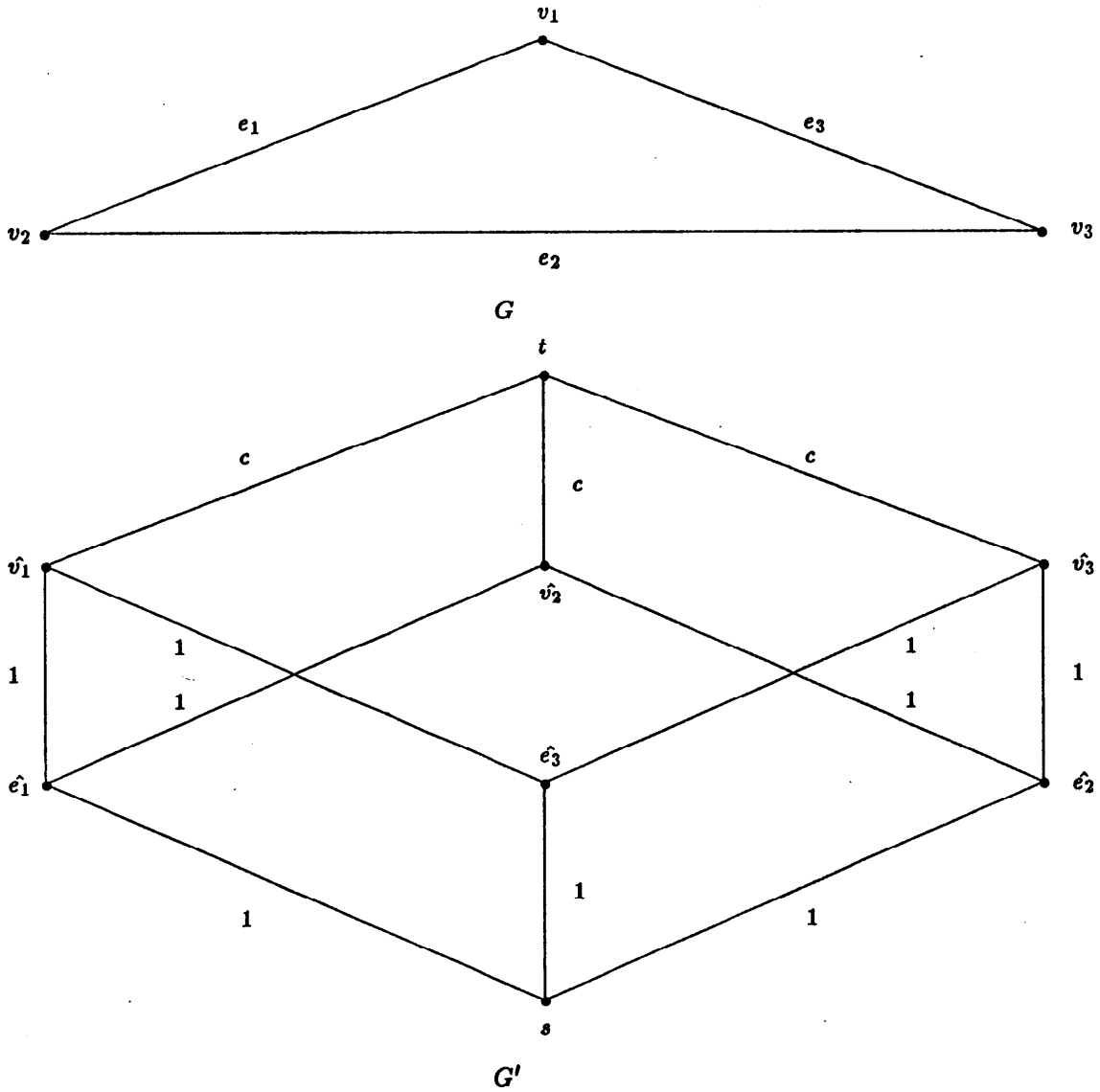

Figure 4.1: Representation to Network Flow Transformation
(All edges are directed upward.)

Suppose that the adjacency lists A_v for $v \in V$ constitute a c-representation of G. Then define the flow f by:

$f(\{s, \hat{e}\}) = 1$ for all $\hat{e} \in V'$ corresponding to $e \in E$;

$f(\{\hat{v}, t\}) = |A_v|$ for all $\hat{v} \in V'$ corresponding to $v \in V$;

finally, for $\hat{e} \in V'$ corresponding to $\{u, v\} \in E$, let

$$f(\{\hat{e}, \hat{u}\}) = \begin{cases} 1, & \text{if } \{u, v\} \in A_v; \\ 0, & \text{if } \{u, v\} \in A_u. \end{cases}$$

It is easy to check that f is a legal flow and that f has total flow $|E|$ and hence is maximum.

Suppose on the other hand that f is a maximum flow with total flow $|E|$. Then we can find adjacency lists A, from f as follows. Let e = $\{u, v\}$ be in E. Place e in A, *if* $f(\{\hat{e}, \hat{u}\})$ = 1. (Note that one of $f(\{\hat{e}, \hat{u}\})$ and $f(\{\hat{e}, \hat{v}\})$ 1has value zero, the other has value one.) ∎

Thus we have established that the c-representation problem can be reduced to the network flow problem. Notice that since we may assume that $|V| \leq |E| \leq c\,|V|$, the size of the network flow problem exceeds the size of the representation problem by at most a factor of O(c). Note also that the reduction can be accomplished in log space.

We next demonstrate that the degree-bounded perfect bipartite matching problem can be reduced to the representation problem. Suppose that G = $(U \cup V, E)$ is a bipartite graph of maximum degree c, and that we wish to find within it a perfect matching. In order to reduce this problem to a representation problem, we introduce the notion of an edge source.

Suppose that a graph H contains as a subgraph K the complete graph on $2c + 1$ vertices. Note that K is c-sparse and thus by Lemma 4.1 it has a c-representation. But since K has exactly the maximum number of edges $(c(2c + 1))$ possessed by any c-sparse graph on $2c + 1$ vertices, each adjacency list in any c-representation of K is full. Therefore, in any c-representation of H, any edge joining a vertex in K to a vertex in $H - K$ must be stored in the adjacency list corresponding to its endpoint in $H -$ K. For this reason, we call the complete graph on $2c$ + .l vertices an edge source.

We transform the bipartite graph G into a graph G' as follows. G' will contain as a subgraph G itself. Now the idea is to arrange things so that in any c-rcprcscntation of G' the adjacency list for any vertex u $\in U$ will have room for exactly one edge from E. For each vertex v $\in V$, the construction will allow room for exactly $d - 1$ edges of E, where d is the degree of v in G. These conditions will ensure that G has a perfect matching if and only if G' has a c-representation. Indeed, an edge {u, v} will be stored in A, if it is a matched edge and will bc stored in A, otherwise.

Thus to construct G', WC first add to G an edge source K, i.e. a complete graph on $2c + 1$ vertices. For each vertex u $\in U$, WC add c -- 1 edges connecting u to any c $-$ 1 vertices of K. (Since in any c-representation of G' these edges cannot be stored at their K endpoints, they must all be stored at u; this ensures that the adjacency list for u has room for exactly one edge of E.) Likcwisc, let v be in V and let d be the degree of v in G. Add edges connecting v to any c $-$ d + 1 vertices of K. This ensures fulfillment of the second requirement, namely that each v $\in V$ has room for exactly d --- 1 ctlgcs of E.

This construction establishes the following theorem.

Theorem 4.4: G has a perfect matching if and only if G' has a c-representation. ∎

Note that G' contains $|U| + |V| + (2c + 1)$ vertices and at most $|E|$ + (c $-$ 1) $|U|$ + $c\,|V|$ + $c(2c + 1)$ edge. Thus again the size of G' exceeds that of G by at most a factor of O(c) aucl the transformation process requires only log space.

We have established that the sequential time complexity of finding c-representations lies between that of finding perfect matchings in degree-bounded bipartite graphs and that of finding maximum flows. Because the above reductions require only log space, the same statement holds for parallel time complexity in the PRAM model [F-W]. Furthermore, these reductions can be easily performed at negligible cost in the parallel environments of concern here, namely those permitting moderate area and moderate time. Thus we make the somewhat imprecise statement that finding efficient parallel algorithms for the c-representation problem appears to be as difficult as finding good parallel algorithms for the bipartite matching problem.

It is perhaps worth observing that the $2c$-representation problem can be viewed as an easily computed approximation to the c-representation problem. In fact it is possible to construct a poly-log time PRAM algorithm for computing $2c$-representations. Such an algorithm can be based upon the idea of repeatedly removing from the graph all vertices of degree at most $2c$. The following lemma establishes that each such iteration removes at least a fixed fraction of the remaining vertices, and thus O(log n) iterations suffice. This idea is exploited (and explained in greater detail) in Chapter 5 to obtain a two-stage funnelled-pipeline algorithm for computing $2c$-representations.

Lemma 4.1: Every c-sparse graph on n vertices has at least $\frac{n}{2c+1}$ vertices of degree at most $2c$.

Proof 4.3: Let V_B be the set of vertices of degree greater than $2c$; let V_S contain those of degree at most $2c$. Let E_B be the set of edges with both endpoints in V_B, let E_S be the set of edges with both endpoints in V_S, and let E_X contain those edges joining a vertex in E_S to one in E_B. By the definition of E_B, we have $(2c+1)|V_B| \leq 2|E_B| + |E_X|$. By the definition of sparseness, $|E_B| \leq c|V_B|$. By the definition of V_S, we have $|E_X| \leq 2c|V_S|$. Thus $(2c + 1)|V_B| \leq 2c|V_B| + 2c|V_S| = 2cn$. Hence $|V_B| \leq 2cn/(2c + 1)$ and $|V_S| \geq n/(2c + 1)$. I

We remark that the subject of poly-log time PRAM approximation algorithms for finding solutions to difficult problems (P-Complctc problems) is explored in [AM]. We also note the existence of a probabilistic poly-log time PRAM algorithm for solving the matching problem [KTJW]. In [CSV] it is demonstrated that the network flow problem (where the capacities are specified in unary) can be reduced to bipartite perfect matching. Thus there is a probabilistic poly-log time PRAM algorithm for finding c-representations. However, WC do not know whether there is a deterministic poly-log time PRAM algorithm for finding c-representations (or, for that matter, finding Xc-rcprcscntations for any $\lambda < 2$).

4.5 Conclusion

WC have used the funnelled-pipeline paradigm to develop an efficient parallel solution to the problem of converting the adjacency matrix of a sparse graph into an equivalent set

of adjacency lists. One application of this data-rearrangement algorithm is to the problem of determining graph planarity.

We have also shown that the representation problem for sparse graphs is closely related in complexity and structure to several other well-known graph problems. This relationship gives rise to a number of open questions. Among those questions are the following.

1a. Are there area- and time-efficient parallel algorithms for solving the AC-representation problem for $\lambda < 2$?

b. Are there fast deterministic PRAM algorithms for solving the Xc-representation problem for $\lambda < 2$?

2a. Are there area- and time-efficient parallel algorithms for solving the bipartite matching problem?

b. Are there fast deterministic PRAM algorithms for solving the bipartite matching problem?

3. Are any of these problems log space complete for the class *P*?

5. Two-Stage Funnelled Pipelines

5.1 Introduction

Each of the funnelled-pipeline algorithms presented in previous chapters used a row-by-row input schedule, and an architecture composed of a linear array of processors. In this chapter, we consider an alternative structure. This structure, the two-stage funnelled pipeline, makes use of a more powerful architecture and reads its input in a block-by-block fashion. We attempt to contrast these two methods of pipelined filtration.

5.2 Two.-Stage Architecture

Our previous funnelled-pipeline algorithms required only modest interprocessor communication. This requirement was met by a simple ethernet or bus interconnection network. One advantage of such simple networks is that (in our VLST model) they require very li ttle area to implement. Ilowevcr, several other interconnection schemes arc nearly as space cfficient, but support significantly greater communication. We employ two of these arrangements, namely the mesh of processors and the mesh of trees.

In both of these architccturcs, the n processors arc arranged in a $\sqrt{n}$-by-$\sqrt{n}$ lattice pattern. In the mesh of processors, each unit is connected to its four nearest neighbors. In the mesh of trees, this pattern is augmented by the addition of row and column trees. Thus there is a tree is associated with each row and each column of the mesh. Each such tree has as its leaves the processors in the corresponding row or column. It is convenient to identify the root node of the column-i tree with the root of the row-i tree. Both the mesh-of-processors and the mesh-of-trees architectures can be implemented in O*(n) area. We refer the reader to [U] and [F-L] for further detail.

Filter stage one will execute on a mesh of trees; filter stage two on a mesh of processors. The two stages arc connected together by $\sqrt{n}$ links: the common root of the i^{th} row and column trees of the mesh of trees is connected to processor i of the first column of the mesh of processors. This O^* (n) area layout is illustrated in Figure 5.1.

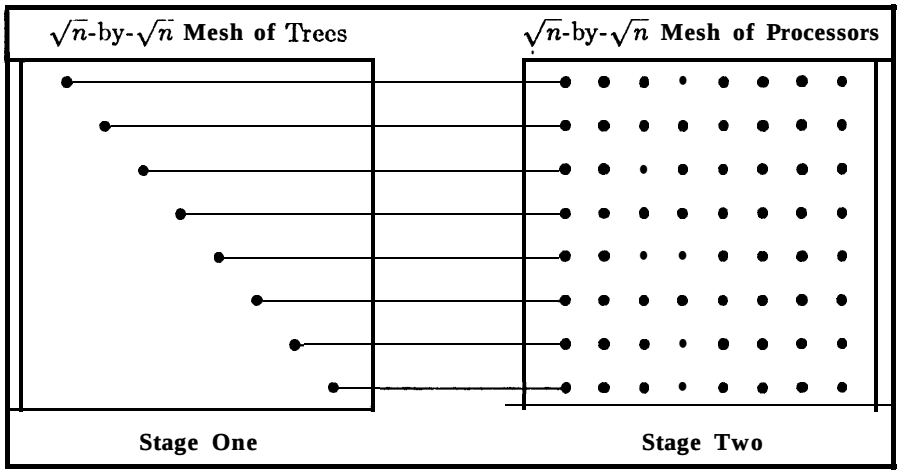

Figure 5.1: Two-Stage Pipeline Architecture.
(Only the tree-root processors are indicated in the mesh of trees)

5.3 A Two-Stage Connected Components Algorithm

We again assume that the connected components problem is input as an n-by-n adjacency matrix A. Instead of reading the matrix row by row, we read it subblock by subblock. Specifically, we partition the matrix A into $\frac{\sqrt{n}}{2}$-by-$\frac{\sqrt{n}}{2}$ subblocks $B_{i,j}$ as follows: $B_{i,j}$ (for $0 \leq i \leq j < 2\sqrt{n}$) contains the entries $A_{(v)}$ for $i\frac{\sqrt{n}}{2} \leq u < (i+1)\frac{\sqrt{n}}{2}$ and $j\frac{\sqrt{n}}{2} \leq v < (j+1)\frac{\sqrt{n}}{2}$. This partitions A into $2n + \sqrt{n} = O(n)$ subblocks. Each of these subblocks represents a subgraph of the input containing at most $\sqrt{n}$ vertices (each "diagonal" block $B_{j,j}$ contains only $\frac{\sqrt{n}}{2}$ distinct vertices; all others contain $\sqrt{n}$ vertices).

The stage-one filtration consists of reading a subblock $B_{i,j}$ into the mesh of trees, computing the connected components of the graph represented by $B_{i,j}$, and passing the result to the stage-two processor. We examine each of these steps in turn.

It is most convenient to read the bits of subblock $B_{i,j}$ into the processors so as to maintain an adjacency matrix format; we let row m and column m of the processor array correspond to vertex $i\frac{\sqrt{n}}{2} + m$ if $m < \frac{\sqrt{n}}{2}$ and to vertex $j\frac{\sqrt{n}}{2} + (m - \frac{\sqrt{n}}{2})$ if $m \geq \frac{\sqrt{n}}{2}$ (for diagonal blocks, *i.e.* if i = j, we ignore all but the first $\frac{\sqrt{n}}{2}$ rows and columns of the $\sqrt{n}$-by-$\sqrt{n}$ processor array).

The connected components of the graph represented by $B_{i,j}$ can be computed in $0^*(1)$ time on the mesh of trees. Such speed is possible because there are only $\sqrt{n}$ vertices (but n processors). The idea is to merge repeatedly groups of vertices belonging to the same connected components. In each of $O(\log n)$ iterations, each group selects an adjacent lower-numbered group (if any exist) and merges with it. Each iteration can be accomplished in $0^*(1)$ time. A full presentation of the method is given in [U]. At the conclusion of this process, each vertex v, represented by the root of its row and column trees, knows its connected component number cc(v). This data is recorded by having each vertex v generate

an edge from itself to $cc(v)$. It is then possible to transfer in parallel these $\sqrt{n}$ edges to the stage-two filter.

We should note that the algorithm that we are describing is not a faithful filtration. However, it is not difficult to augment the stage-one filtration so that it finds a spanning tree. We leave the details to the reader.

The second-stage filter serves partly as a buffer. Each of its processors (u, v) has two registers: $buffer_{u,v}$ and $edge_{u,v}$. Each is capable of holding an edge; thus the total capacity of stage two is $2n$ edges. Each processor uses its *buffer* register for temporarily holding stage-one output. After each activation of stage one, the data buffered in each column of the stage-two filter is shifted right by one column to make room for the next column of stage-one output. Thus, at least for buffering purposes, the rows of the stage-two filter array serve as nothing more than shift registers. After $\sqrt{n}$ successive activations of stage one, the stage-two buffer is full.

In order to empty the buffer, a connected components computation is performed on the graph represented by the edges $E = \{ buffer_{u,v} \mid 0 \leq u, v < \sqrt{n} \} \cup \{ edge_{u,v} \mid 0 \leq u, v < \sqrt{n} \}$. It should be pointed out that all registers $edge_{u,v}$ are empty the first time that the buffer becomes full. The stage-two connected components computation results in a set of at most $n - 1$ edges representing the connected components of E. These edges arc stored back in the *edge* registers. Thus the buffer registers are emptied, and stage two will again be ready to receive another n edges from stage one.

Note that the set of edges E may comprise a graph on all n vertices. Nevertheless, it is possible to compute the connected components of E in $O^*(n^{1/2})$ time. Such an algorithm can be based on the fact that it is possible to sort O(n) records in $O^*(n^{1/2})$ time on a mesh of n processors (refer to [T-K]). Connected components are again computed by repeatedly merging groups of vertices. Each vertex is represented not by a particular processor, but by a record. Sorting is used to find the lowest numbered adjacent group. It is also used for "message routing" (both the one-to-one routing operation and the one-to-many routing operation are used in the connected components algorithm). We lcave the details of this connected components algorithm as an interesting cxcrcisc for the reader.

The outline of the two-stage-pipeline connected components algorithm is as follows.

```
program components;
begin
  Initialize;
  for bigcycle := 0 to 2√n + 1 do
    for littlecycle := 0 to √n − 1 do
      Read a subblock into stage-one processors;
      Compute the connected components of the subblock;
      Shift the resulting connected components output into stage two
      od;
    co Note that above loop takes time O*(n^1/2);
    co Stage-two buffer is now full. Empty it oc;
    Find the connected components of the at most 2n stored edges stored in time O*(n^1/2)
    od;
  Output stage-two edge registers
end components .
```

Because the speeds of the two filter stages are properly matched, the entire algorithm takes time O*(n) to compute the connected components of the input graph. Thus we have yet another when- and where-determinate 0* (n)-area-and-time connected components algorithm.

We should note that this algorithm differs from the previous ones in that it requires input pads to lie in the interior of the circuit. In contrast, the previous algorithms can all be implemented as "boundary layouts".

5.4 Finding Minimum Spanning Trees and 2c-Representations

The organization used to implement our two-stage connected components algorithm applies equally well to many closely related problems such as finding spanning trees and minimum spanning -trees. Again we leave the details to the reader.

We illustrate a two-stage funnelled-pipeline 0* (n)-area-and-time algorithm for finding %-representations of c-sparse graphs. The operation of our algorithm consists of two major steps. The first step, the read-and-store step, uses a two-stage funnelled-pipeline algorithm to read and store all edges of the input graph. The second step rearranges these stored edges so as to produce a valid $2c$-representation of the entire graph.

The read-and-store step itself makes use of a $2c$-representation algorithm to ensure the orderly transfer and storage of input edges. Every time a subblock of the adjacency matrix is read, its $2c$-representation is computed by the stage-one filter. This permits the stage-one filter to transfer rapidly these edges to the stage-two buffer; each of the $\sqrt{n}$ root processors transfers its at most $2c$ edges to the corrcspondiug row of the stage-two mesh. After every $\sqrt{n}$ repetitions of this process, it becomes necessary to "compact" the stage-two buffer. This requires nothing more than a sorting operation to rearrange the at most cn stored edges.

Both the stage-one filtration and the final $2c$-representation computation are performed by means of a "grcctly" algorithm. In Lemma 4.1 we remarked that every c-sparse graph on n vcrticcs has at least $\frac{n}{2c+1}$ vcrticcs of degree at most $2c$. This observation suggests a very simple algorithm. The idea is to identify those vertices with degree $2c$ or less. All such vortices then grab their incident edges and remove them from the graph. This process, which requires O*(c) time on a mesh of trees, and $O^*(\sqrt{n})$ time on a mesh of processors, eliminates at least a fraction $\frac{1}{2c+1}$ of the vcrticcs. By repeating this process $O(\log n/\log \frac{2c+1}{2c})$ times, a $2c$-representation is found. Thus each first-stage filtration requires time 0 * (1) and the final $2c$-representation computation requires time 0* $(\sqrt{n})$. Note that the final $2c$-representation computation is faster than it need be; any O*(n) time procedure would suffice.

5.5 Conclusion

We have seen that many of our graph problems can be solved efficiently with two-stage funnelled pipelines. Thus there are good algorithms for these problems both with row-by-row, and with subblock-by-subblock, input schedules. We note several open yucstions.

The two-stage funnelled pipeline seems to provide greater computational power than that provided by census function architectures. Thus it seems reasonable to expect that there are natural graph problems that can only be satisfactorily solved in the two-stage model. On the other hand, our row-by-row algorithms made effective use of the constraint on the number of "leaders" appearing in subgraphs. This constraint made possible certain types of rapid filtration. In particular, it is not clear whether there is an efficient two-stage funnelled-pipeline algorithm for solving the biconnected components problem. Thus we leave the reader with the following questions.

1. Is there a two-stage funnelled-pipeline algorithm for solving the biconnectcd components problem?
2. Find examples of (uncontrived) graph problems that are solvable with the subblock-by-subblock input schedule, but are not solvable with row-by-row schedules.

6. Strongly Connected Components

6.1 Introduction

In this section we consider the strongly connected components problem. This problem is inherently more difficult than those examined in previous chapters. In fact, any when- and where-determinate circuit that solves the strongly connected components problem requires $\Omega(n^2)$ area. This is because there is no way to safely discard edges when examining only a portion of the input. In other words, no filter can significantly reduce the volume of data. It should be noted that we are requiring the input schedule to be scmelcctive; that is, the data is read only once. We will mention other classical problems that share this difficulty.

It is possible however to obtain an efficient strongly connected components algorithm by relaxing the requirement of when- and where-determinacy. We will develop such an algorithm, making use of techniques similar to those presented in previous chapters.

6.2 The, Strongly Connected Components Problem

Let $G = (V, E)$ bc a directed graph. We define a relation $\star$ on V as follows: $u \star$ v if $u =$ v or if there exist directed paths in E from u to v and from v to u. It is easy to sec that $\star$ is an equivalence relation. The strongly *connected* components of G are defined to bc the equivalence classes of $\star$. They arc rcprcscntcd by a function $scc : V \longrightarrow V$ where $scc(v)$ is the lowest-numbered vertex lying in the same strongly connected component as v.

As usual, we will assume that the input is presented as au n-by-n adjacency matrix A. Entry $A_i(j) = 1$ if there is an edge directed from vertex i to vertex j. We insist that the circuit produce as output the function scc.

6.3 A Lower Bound

We present a lower bound of $\Omega(n^2)$ on the area required for computing strongly connected components. This lower bound depends upon some assumptions regarding input and output conventions. For our purposes, it would suffice to prove this bound in the case of when- and where-determinate circuits that read the input data once. However, it is not much more difficult to prove that the lower bound applies to a much larger class of input-output schedules. We show that the bound applies to any circuit in which the inputs arc read in any fixed sequcncc. WC call such input schedules what-determinate.

Definition 6.1: A circuit is what-determinate provided that its inputs $I_1, I_2, \ldots, I_k$ are read once in some fixed sequence.

Definition 6.2: Let $I = \{I_1, \ldots, I_k\}$ denote the set of input variables of a what-determinate circuit. Then there exists an *input schedule* S associated with the circuit, where $S(t)$ denotes the set of variables read at time t. Note that $S(t) \cap S(t') = \emptyset$ when $t \neq t'$ and that $\bigcup_t S(t) = I$.

We prove our bound by means of an adversary argument. In particular, we will show the existence of a large "fooling set" of subgraphs defined by inputs read early in the schedule. Because the circuit will have to remember which of these graphs it has encountered, it will need a large amount of memory, and hence, be large.

Assume that S is the input schedule associated with some what-determinate circuit computing strongly connected components on n-vertex graphs represented by adjacency matrices. Thus each set $S(t)$ consists of adjacency-matrix elcmcnts. We may assume that for all t we have $|S(t)| < \epsilon n^2$ (where ϵ is any small fixed constant). Otherwise, since each input pad takes constant area, we immediately have $\Omega(n^2)$ area.

Arbitrarily partition the set of vertices into four sets V_1, V_2, V_3 and V_4, each of cardinality $\frac{n}{4}$. It is convenient to view the input variables in $S(t)$ as a set of potentially present edges, namely the set $\{(i, j) \mid$ the variable $A_i(j)$ is in $S(t)\}$. Thus the statement that the edge (i, j) is read at time t means that at time t the bit $A;(j)$ of the adjacency matrix is read, i.e. that the variable $A_i(j)$ is in $S(t)$. Let t_0 be the earliest time by which half of the edges directed from some V_i to some other $V_{i'}$ have been read. That is, let t_0 be the minimum such that there exist i and i' with $\left|\bigcup_{t \leq t_0} S(t) \cap (V_i \times V_{i'})\right| \geq \frac{1}{2} |V_i \times V_{i'}|$. We assume without loss of generality that $i = 1$ and $i' = 2$. Let $E = \bigcup_{t \leq t_0} S(t) \cap (V_1 \times V_2)$ be the set of "early" cdgcs. Thus $|E| \geq \frac{1}{2} |V_1 \times V_2| = n^2/32$. Our fooling set will consist of graphs which differ from each other only in their intersection with E.

Let the "late" edges, L, be the set $\bigcup_{t > t_0} S(t) \cap (V_4 \times V_3)$. The adversary will make use of the cdgcs in L in order to distinguish bctwccn members of the fooling set. Note that L is quite large. Because t_0 was chosen to be minimum, we have that $\left|\bigcup_{t < t_0} S(t) \cap (V_4 \times V_3)\right| < \frac{1}{2} |V_4 \times V_3|$. Hence, $\left|\bigcup_{t \geq t_0} S(t) \cap (V_4 \times V_3)\right| \geq \frac{1}{2} |V_4 \times V_3|$ and thus $|L| \geq \frac{1}{2} |V_4 \times V_3| - \epsilon n^2 = (1 - \epsilon') \frac{n^2}{32}$.

We will pair each vertex in V_1 with a mate in V_3. Similarly, each vertex in V_2 will be mated with one in V_4. Such a pairing will be denoted by a function P which bijectivcly maps V_1 to V_3 and V_2 to V_4. Each graph in our fooling set will consist of the set of cdgcs $\{(v, P(v)) \mid v \in V_2\} \cup \{(P(v), v) \mid v \in V_1\}$ (in addition to some edges in E). Observe that no such graph contains any directed cycles, and thus each vertex forms its own strongly connected component. Suppose, however, that the edge (v_1, v_2) for $v_1 \in V_1$ and $v_2 \in V_2$ were present. If the adversary can install the edge $(P(v_2), P(v_1))$, then a unique cycle is formed, and the strongly connected component structure is altered accordingly. Of course, if the edge (v_1, v_2) were not prcscnt, then the edge $(P(v_2), P(v_1))$ would create no cycle. So, if the edge $(P(v_2), P(v_1))$ lies in L, the adversary can use it to test for the presence of the edge (v_1, v_2).

Definition 6.3: Suppose that P is a pairing. The edge (v_1, v_2) is said to be *unconcealed* if $(v_1, v_2) \in E$ and $(P(v_2), P(v_1)) \in L$.

Our objective is to find a pairing P that results in a large set of unconcealed edges.

Definition 6.4: If P is a pairing, let $K(P) = |\{e \mid e \text{ is unconcealed under } P\}|$.

Lemma 6.1: There exists a P such that $K(P) = \Omega(n^2)$.

Proof 6.1: Let $K = \sum_P K(P)$. Let m be the number of possible pairings; thus $m = \left(\left(\frac{n}{4}\right)!\right)^2$. For $e \in E$ let $\hat{K}(e) = |\{P \mid e \text{ is unconcealed under } P\}|$. Note that by interchanging summation order, we also have $K = \sum_{e \in E} J?(e)$. But for all $e \in E$, we have $\hat{K}(e) = |L| \left(\left(\frac{n}{4} - 1\right)!\right)^2$ (for a given $e' \in L$, there are $\left(\left(\frac{n}{4} - 1\right)!\right)^2$ pairings under which e' can be used to test for the presence of e). Thus $K = |E|\,|L|\left(\left(\frac{n}{4} - 1\right)!\right)^2$. Hence there is a P for which $K(P) \geq \frac{K}{m} = |E|\,|L|\left(\frac{n}{4}\right)^{-2} = \Omega(n^2)$. ∎

Let P be such that K(P) = $\Omega(n^2)$ and let $E' \subseteq E$ be the set of unconcealed edges. Let $\hat{P}$ be the pairing edges derived from P as described above. Then let the fooling set F consist of all graphs of the form $\hat{E} \cup \hat{P}$ for $\hat{E} \subseteq E'$. Note that $|F| = 2^{\Omega(n^2)}$.

We claim that the circuit cannot operate correctly if any two distinct members f_1 and f_2 of F result in the same circuit state at time t_0. This claim is established by contradiction; suppose that f_1 and f_2 result in the same circuit state at time t_0. Choose some e in the symmetric difference between f_1 and f_2. By construction, e is unconcealed. Hence the adversary can USC some edge e' of L (which is read only after time to> to test for e. The strongly connected components of $f_1 \cup \{e\}$ differ from those of $f_2 \cup \{e\}$. In particular, the value taken by the function *scc* on at least one endpoint of e depends on whether the input graph is $f_1 \cup \{e\}$ or $f_2 \cup \{e\}$. Therefore the circuit cannot have output this value before time t_0. Furthermore, since by assumption the circuit arrives in the same state at time t_0 for both f_1 and f_2, and since the inputs read after t_0 are the same in either case, the computation sequence after t_0 (and hence the output) is the same in either case. The resulting contradiction proves that the circuit must have $2^{\Omega(n^2)}$ possible states. Therefore $\Omega(n^2)$ is a lower bound on the circuit area.

Theorem 6. 1: Any what-determinate circuit computing strongly connected components requires $\Omega(n^2)$ area. ∎

6.4 A Strongly Connected Components Algorithm

We now describe the consequences of relaxing our requirement of what-determinacy. In [L-V] an $O^*(n)$-area-and-time circuit is presented Chat computes strongly connected components. Their circuit both reads the input twice and makes USC of a data-dependent input schedule. Techniques similar to our filtering methods can be used to improve the result.

Our circuit accepts as input an n-by-n matrix *A*. Entry $A_i(j) = 1$ if there is an arc directed from vertex i to vertex j. The rows are read one at a time, but in an order that depends on the data. However the data is read once and the timing of successive row reads is data independent; each row takes O*(l) time to process.

The architecture consists of n simple processors. They arc interconnected by an ethernet (or a binary tree). As mentioned in Section 2.4 we make use of the Broadcast, Maximum and Minimum operations. Each processor corresponds to a vertex of the graph. The correspondence is permanent; processor i is responsible for vertex i.

The algorithm presented here is, in essence, a parallel implementation of the well known depth-first search method for determining strongly connected components [AHU]. At each stage, a "current" vertex, c, is selected according to a depth-first ordering. The adjacency matrix row for this vertex (*i.e.* the row containing the edges departing from c) is read into the processors. This data is then processed so that the state of the processor registers correctly reflects both the strongly connected components induced by the rows so-far read, and the information needed for processing the, as yet, unread rows.

Each processor i contains the following registers:

(1) status(i): The possible values and their meanings are as follows:
- *untouched*: No edge directed towards vertex i has yet been encountered.
- *pending:* A "tentacle" edge directed towards vertex i has been encountered, but this vertex has yet to be explored.
- closed: Vertex i and its descendants have been explored, and the strongly connected component containing vertex i has been determined.
- *open:* **Vertex i has been explored, but the algorithm is still examining its descendants.**

(2) $dfno(i)$: This rcgistcr stores the depth-first number of vertex i (*i.e.* the time at which vcrtcx i is first visited by the depth-first search). It is of significance only when status(i) is either open or *closed*.

(3) $timenc(i)$: This Geld is used to enforce depth-first search. It has meaning only for pending vertices. *timenc* contains the depth-first number of the last encountered vertex with a "tentacle edge" to vertex i.

(4) $scc\_dfno(i)$: This field contains the lowest depth-first number of any vertex known to be in the same strongly connected component as vertex i. This field serves roughly the same purpose as LOWLINK in the strongly connected components algorithm presented in [AI IU].

There arc two significant "global" rogistcrs. The rcgistcr *time* counts the nu mber of vertices that have been explored. Thus it, contains the depth-first number of the current vertex. The register c contains the name of the current vertex.

The following facts are central to understanding the algorithm.

(1) A strongly connected component is identified by the lowest depth-first number of any vertex it contains. The vertex with lowest depth-first number is called the *root* of the strongly connected component.

(2) Vertices arc explored in depth-first order. It is important to note that once a depth-first search explores the first vcrtcx of a given strongly connected component, it does not

backtrack from that vertex until it has explored all vertices in that strongly connected component.

(3) After each stage (*i.e.* after a vertex is explored), the fields *scc_dfno* correctly reflect the strongly connected components induced by the so-far read edges.

(4) All strongly connected components containing *closed* vertices, contain only *closed* vertices. Furthermore, such components are complete even with respect to unexamined edges (*i.e.* they contain everything they will ever contain).

(5) All strongly connected components containing *open* vertices, contain only *open* vertices (however such components are not yet necessarily complete).

Note that (4) and (5) justify the use of the terms *open* and *closed* components.

(6) The *open* strongly connected components form a linear chain ordered by ancestry; this order is compatible with the depth-first numbering of the components' roots. Thus if vertices i and j are *open* and $scc\_dfno(i) < scc\_dfno(j)$ then the root of i's strongly connected component is an ancestor of the root of j's strongly connected component. In particular, the root of every *open* component is an ancestor of the current vertex.

The edges emanating from the current vertex are classified into three categories:

- An edge is called a tentacle *edge* if it is directed from the current vertex c to any vertex *j* where *status(j)* = *untouched* or *status(j)* = *pending* (*i.e.* if j is unexplored). If *status(j)* = *untouched*, this tentacle edge is "remembered" by storing *time (i.e.* the depth-first number, of c) in *timenc*(*j)* and setting *status(j)* to *pending.* This permits the algorithm to perform a depth-first search. However, if *status(j)* = *pending* there must have already been encountered a tentacle edge from some vertex r to j. We claim that this previous tentacle edge to j can be safely forgotten: r must already have been explored since depth-first search is employed, and by invariant (4), *r* cannot be *closed.* Hence *r* must be *open* and invariant (6) guarantces that r is an ancestor of c. Therefore any path using the tentacle edge from r to j can be rerouted from *r* to c and then from c to j by the newly discovered tentacle. Thus the previous tentacle edge is irrelevant to the determination of the strongly connected components. So the algorithm simply sets *timenc* (j) to *time,* thereby remembering only the latest tentacle edge.
- An edge *(c, j)* is a *closed cross edge* if *status(j)* = *closed.* Invariant (4) implies that closed cross edges can bc ignored.
- An edge (c, j) is an *open edge* if *status(j)* = *open.* Note that this category includes, among others, what are called back edges in [AHU]. Since, by invariant (6), the root of j's strongly connected component is an ancestor of c, it must be the case that c and j belong in the same strongly connected component. In fact, invariant (6) implies that all open components with depth-first number greater than or equal to $scc\_dfno(j)$ must be merged. Note that at each stage the algorithm need only consider one open edge, namely the open edge which leads to the vertex *k* with lowest *scc_dfno.* By merging all open components with depth-first number greater than or equal *Co* $scc\_dfno(k)$ (into a single component with depth-first number $scc\_dfno(k)$), the algorithm correctly disposes of all open edges and maintains invariant (3).

The subroutine below explores the current vertex c. The routine for selecting the current vertex is given later.

```
procedure explore;
begin
  co Visit the current vertex c oc;
  co read c's input row and initialize processor registers oc;
  for all k, 0 ≤ k < n do in parallel read adjacency matrix bit A,(k) into edge(k) od;
  status(c) := open;
  dfno(c) := time;
  scc_dfno (c) := time;
  co Deal with tentacles hanging from vertex c oc;
  for all i, 0 ≤ i < n such that status(i) ∈ {untouched,pending} and edge(i) = 1 do in
  parallel
    status(i) := pending;
    timenc(i) := time
    od;
  co Process open edges oc;
  if |{i | 0 ≤ i < n and status(i) = open and edge(i) = 1}| > 0 then
    let t be such that scc_dfno(t) = min{scc_dfno(i) | status(i) = open and edge(i) = 1};
    co merge all open components whose dfs number is at least t oc;
    for all i, 0 ≤ i < n such that status(i) = open and t ≤ scc_dfno(i) do in parallel
      see-dfno (i) := t od
    fi;
end explore;
```

Note that this procedure maintains invariants (1), (3), (5) and (6). Note also that it requires $O^*(1)$ time on an ethernet architecture.

The remaining issue is performing the depth-first search and placing vertices in *closed* status. Depth first search (invariant (2)) is accomplished by selecting the new current vertex m arbitrarily from the set of *pending* vertices with largest *timenc* fields. (If no vertices are *pending,* the algorithm arbitrarily selects an untouched vertex.)

There arc two cases to consider. If $timenc(m) = time$, then m is a descendant of the current vertex and no special action is required. On the other hand, if $timenc(m) < time$, the depth-first search is backtracking. In this case it may bc necessary to close some components (in order to maintain invariants (4), (5) and (6)). Now the latest encountered tentacle to m comes from some *open* vertex j and $dfno(j) = timenc(m)$. Thcreforc all (if any) *open* components with depth-first number strictly greater than $scc\_dfno(j)$ must be closed. By invariant (2) these components cannot be ancestors of m, and all other *open* components are ancestors of m. Note that invariant (6) is maintained. The procedure is given below.

```
procedure nextcurrent;
begin
  co pick the next current vertex oc;
  if |{i | 0 ≤ i < n and status(i) = pending}| > 0 then
      Pick m so that timenc(m) = max{timenc(i) | 0 ≤ i < n and status(i) = pending};
      t := timenc(m)
    else
      co start traversing a new tree oc;
      let m be any untouched vertex;
      t := -1
    fi;
  co close components that are now complete oc;
  for all i, 0 ≤ i < n such that status(i) = open and t < scc_dfno(i) do in parallel
    status(i) := closed od;
  co set current vertex register oc;
  time := time + 1;
  c := m;
end nextcurrent; __
```

Note that this procedure requires O*(l) time. The main program is as follows:

```
program SCC;
  begin
  co initialize oc;
  for all i, 0 ≤ i < n do in parallel status(i) := untouched od;
  co perform depth-first search oc;
  repeat n times
    call nextcurrent;
    call explore
    od;.
  co write the output oc;
  for all i, 0 ≤ i < n do in parallel
    write scc_dfno(i) od;
end.
```

This completes our one-pass, 0* (n)-area-and-time, data-dcpendcnt-input-schcdulc,' strongly connected components algorithm.

6.5 Other Hard Problems

We have seen that the strongly connected components problem is inherently more difficult than those examined in previous chapters; any what-determinate circuit for computing strongly connected components requires a large amount of memory. A number of other important graph problems share this unattractive property. We list some of these below.

(1) Breadth-first search on undirected graphs.
(2) Breadth-first search on directed graphs.
(3) Depth-first search on directed graphs.
(4) Matching.

All of these problems require $\Omega(n^2)$ area if they are to be solved by what-determinate circuits. Because the formal proofs are somewhat tedious, we merely give the main ideas.

The $\Omega(n^2)$ lower bound is easily established for problems one through three if the root of the traversal is not data dependent; for instance, if vertex zero is always the first vertex visited. In this case, (which we call the fixed-root case) $2^{\Omega(n^2)}$-sized fooling sets are readily constructed. Matters are slightly more complicated when the circuit is allowed to choose its initial root according to the data. We call this version the floating-root problem. Perhaps the easiest way to proceed is to observe that a fixed-root problem can be simulated by a corresponding floating-root problem. The idea for breadth-first search on undirected graphs is to make two copies of the fixed-root problem and join them in an appropriate manner. One satisfactory way to join the copies is to identify the two instances of the desired root vertex. This will ensure that at least one copy is traversed starting at the desired root. Observe also that directed breadth-first search simulates undirected breadth-first search. The bloating-root depth-first search can also be handled by the two-copy idea; WC leave to the reader the problem of finding a suitable way to join the copies.

The lower bound for matching can bc established in much the same way as that for strongly connected components. One must show the cxistcncc of two equal-sized sets of vertices V_1 and V_2, a pairing P between them, and two "probe" vcrticcs p_1 and p_2 with the following properties.

- There is a set E of $\Omega(n^2)$ cdges joining vertices in V_1, and all adjacency-matrix bits corresponding to members of E are read prior to time t_0.
- The set of adjacency-matrix bits corresponding to the edges in $L = \{\{i, j\} \mid i \in \{p_1, p_2\}$ and $j \in V_2\}$ arc read after time t_0.

The fooling set consists of all graphs of the form $\hat{E} \cup \hat{P}$ where $\hat{E} \subset E$ and $\hat{P} = \{\{i, P(i)\} \mid i \in V_1\}$. The adversary probes for an edge $\{u, v\} \in E$ by introducing the edges $\{p_1, Y(U)\}$ and $\{p_2, P(v)\}$. Then there is a perfect matching among the vcrticcs in $V_1 \cup V_2 \cup \{p_1, p_2\}$ if and only if the edge $\{u, v\}$ is prcscnt.

It is a surprising fact that the problem of performing depth-first search on undirected graphs can bc solved with much less memory than any of the above problems. Richard Andcrson has dcvclopcd an $O^*(n^{5/3})$-area when- and where-determinate circuit that performs depth-first search on n-vertex undirected graphs [A].

7. References

[A] R. Anderson, A Horrendously Complicated Depth-First Search Algorithm, unpublished memorandum, 1984.

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

[A-K] M.J. Atallah and S.R. Kosaraju, Graph Problems on a Mesh-Connected *Processor* Array, Proceedings of the 14th ACM Symposium on Theory of Computation, pp.345–353 (1982).

[A-M] R. Anderson and E.W. Mayr, A P-complete *Problem and* Approximations *to It,* Stanford Computer Science Department Technical Report No. STAN-CS-84-1014, 1984.

[CLC] F.Y. Chin, J. Lam, and I. Chen, *Efficient* Parallel Algorithms *for some* Graph Problems, Comm. of the ACM, vol.25, pp.659–665 (1982).

[CSV] A.K. Chandra, L.J. Stockmeyer and U. Vishkin, Constant Depth Reducibility, IBM Research Report No. RC9548, August 1982.

[D-S] E. Dekel and S. Sahni, Binary *Trees* and *Parallel* Scheduling Algorithms, IEEE Trans. on Computing, vol.C-32, pp.307–315, (1983).

[FL] F.T. Leighton, New *Lower Bound* Techniques *for VLSI,* Proceedings 22nd IEEE Symp. on Foundations of Computer Science, pp.1–12 (1981).

[F-W] S. Fortune and J. Wyllie, Parallelism in *Random* Access *Machines,* Proceedings of the 10th ACM Symposium on Theory of Computing, pp.114–118 (1978).

[H] S.E. Hambrusch, *VLSI Algorithms for the Connected* Components *Problem,* Penn. State Department of Computer Scicncc Report CS-81-9, 1981.

[KUW] R.M. Karp, E. Upfal and A. Wigdcrson, *Constructing a Perfect Matching is in Random* NC, to appear, 1984.

[L-S] R.J. Lipton and R. Sedgewick, *Lower Bounds* for *VLSI,* Proceedings of the 13th ACM Symposium on Theory of Computing, pp.300–308 (1981).

[L-V] R. J. Lipton and J. Valdcs, *Census Functions: an Approach to VLSI Upper Bounds,* Proceedings 22nd IEEE Symp. on Foundations of Computer Science, pp. 13-22 (1981).

[M-B] R. Metcalfe and D. Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks,* Comm. of the ACM, vol.19, pp.395404 (1976).

[RND] E.M. Rcingold, J. Nicvcrgclt, and N. Deo, *Combinatorial Algorithms:* Theory and Practice, Prentice-Hall, 1977.

[T-K] C.D. Thompson and H.T. Kung, *Sorting on a Mesh Connected Parallel Computer,* Comm. of the ACM, vol.20, pp.263–271 (1977).

[U] J.D. Ullman, *Computational Aspects of VLSI,* Computer Scicncc Press, 1984.