Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1985

Parallel Algorithms for Bridge- and Bi-Connectivity on Minimum Area Meshes

Susanne E. Hambrusch
Purdue University, seh@cs.purdue.edu

Report Number:
85-506

PARALLEL ALGORITHMS FOR BRIDGE- AND BI-CONNECTIVITY ON MINIMUM AREA MESHES

Susanne E. Hambrusch

CSD-TR-506
February, 1985

TR 506

Parallel Algorithms for Bridge- and Bi-Connectivity on Minimum Area Meshes

Susanne E. Hambrusch

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

January 1985

Abstract

We present parallel algorithms for finding the bridge- and bi-connected components of an undirected graph $G=(V,E)$ with n vertices and e edges on 2-dimensional mesh of size $n^{1/2} \times n^{1/2}$. In conventional parallel models any bridge- and bi-connectivity algorithm requires at least n processing elements, and thus our algorithms run on minimum area networks. Our algorithms find the bridge-connected components in $O(n^{3/2})$ time for both input in the form of an adjacency matrix and in the form of edges. For bi-connectivity we show how achieve $O(n^{3/2})$ time when the input is adjacency matrix form, and $O(e+n^{3/2})$ time when the input is in the form of edges.

Key Words

Parallel computation, analysis of algorithms, bi-connectivity, bridge-connectivity, meshes.

This work was supported in part by the Office of Naval Research Contract N00014-84-K-0502.

1. Introduction

The simple interconnection pattern and the uniform wire length of a mesh of processors appear to make it ideally suited for parallel processing and VLSI computation, and numerous researchers have developed parallel algorithms tailored towards the mesh [AK, GKT, H1, KL, MS1, MS2, TK]. In this paper we present parallel algorithms for finding the bridge- and bi-connected components of an undirected graph $G=(V,E)$, $|V|=n$ and $|E|=e$, on a mesh of size $n^{1/2}\times n^{1/2}$. Since, under conventional assumptions for parallel models any algorithm finding the bridge-, bi-, and connected components requires at least n PE's, our algorithms run on a network of minimum area. Developing algorithms for minimum area networks is both of theoretical and practical interest. Of practical interest because area is an expensive resource, and of theoretical interest because of the algorithm and data movement techniques needed.

The $n^{1/2}\times n^{1/2}$ mesh receives n^2 (resp. e) inputs describing the graph in the form of 'input waves', and the algorithms cannot explicitly store the entire input on the mesh. Thus the actual computation has to begin before all the inputs have been read. Already between the reading of input waves our algorithms determine which inputs are irrelevant and can be discarded, and they incorporate relevant inputs (i.e., inputs that contain new information about the graph) into the data structures used on the mesh. Organizing the individual elements of the data structures so that the necessary data movement can be done fast and without 'collisions' is crucial to the efficiency of our algorithms. We next describe our parallel model and our results.

In our model we assume that every input is read once, every output is generated once, and that every PE contains a constant number of registers of $\log n$ bits each. Thus the mesh has a 'storage capacity' of $O(n\log n)$ bits, while the total length of the input is n^2 (resp. $\Theta(e\log n)$) bits. Observe that numerous problems (e.g., directed graph problems, sorting) cannot be solved on networks with storage capacity less than the length of their input [H1]. We consider algorithms in which the graph G is represented by an adjacency matrix as well as algorithms in which G is represented in the form of edges. In the case of an adjacency matrix, the i-th input wave consists of the i-row of the matrix, and in the case of edges, the i-th input wave consists of n arbitrary edges of G. In the i-th input wave PE_j, $1 \le j \le n$, receives exactly on input (which is either the bit a_{ij} or an edge (x_j, y_j)). Our algorithms receive the input waves in a when-indeterminate mode [U]; i.e., the time at which the i-th input wave is read may depend on the data.

In this paper we show how to find the bridge-connected components (i.e., the maximum subgraphs of G for which the removal of an edge leaves the subgraph connected) in $O(n^{3/2})$ time for both input in the form of an adjacency matrix and edges. Our bridge-connectivity algorithms number the bridge-connected components of the graph, and the output consists, for every vertex, of the number of the bridge-connected component the vertex is in. We show how to determine the bi-connected components (i.e., the removal of a vertex leaves the subgraph connected) in $O(n^{3/2})$ time when the input is in adjacency matrix form, and in $O(e+n^{3/2})$ time when the input is in the form of edges. The bi-connectivity algorithms also number the bi-connected components. The output lists, for every vertex, the bi-connected components containing this vertex. Note that, since bi-connectivity induces an equivalence relation on the edges, a vertex can be in more than on bi-connected component [AHU].

Algorithms for graph problems on parallel models with enough PE's and memory to store a representation of the graph explicitly during the entire computation have been studied extensively for a variety of parallel models [AK, DNS, HCS, JS, NS1, SJ, SV, TC]. The issues involved when only part of the input is available at any time during the algorithm and where this input is processed (i.e., irrelevant inputs are discarded) before the next input wave is read, are quite different. Lipton and Valdes [LV] and Hochschild et al. [HMS] consider binary tree networks with n leaves for solving graph problems with adjacency matrix input. The algorithms in

[HMS] require $\log n$ registers per PE, and the bi-connectivity algorithm in [LV] reads the adjacency matrix twice. Hambrusch [H2] uses the model of this paper and describes algorithms on $O(n)$ area meshes for finding the connected components in $O(n^{3/2})$ time for both forms of input.

2. Bridge-Connectivity

In this section we present an algorithm for finding the bridge-connected components on a 2-dimensional mesh of $O(n)$ area in time $O(n^{3/2})$. We first give the algorithm for input in the form of an adjacency matrix, and then describe the modifications to be done when the graph is represented in the form of edges. In our algorithms we assume that the n PE's, $PE_1, \cdots ,PE_n$, are arranged in snake-like row-major order; i.e., PE_i is directly connected to PE_{i-1} and PE_{i+1}, provided they exist. This assumption is for convenience only, and our time bounds hold when other standard indexing schemas are used. The time bounds of our algorithms are further independent of whether all or only the PE's on the boundary of the mesh can perform I/O. We make the standard assumption that in unit time every PE can perform an operation using its own registers or send the content of some of its registers to an adjacent PE; for further details of the model see [H1].

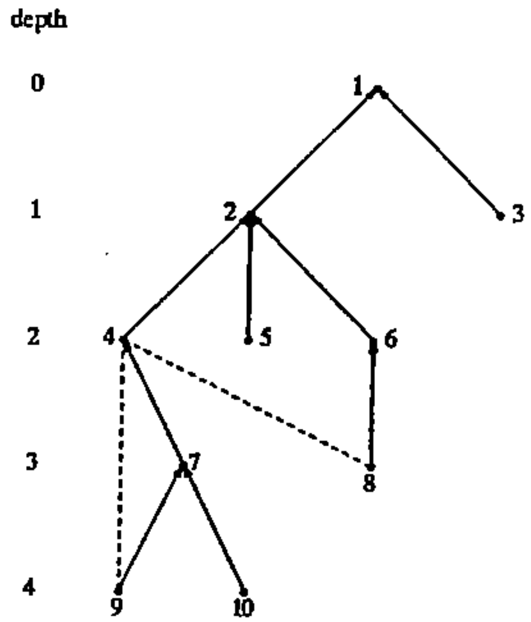
We start with an informal description of the approach used in the bridge-connectivity algorithm. The algorithm processes the i-th input wave (i.e., the i-th row of the adjacency matrix) completely before reading the $(i+1)$-st input wave. Throughout the algorithm vertex i has two integers, C_i, the current component number of i, and B_i, the current bridge-connected component number of i, associated with it, $1 \le i \le n$. Initially, $B_i = C_i = i$, $1 \le i \le n$. These two entries are stored in PE_i in the mesh. The algorithm puts two vertices in the same bridge-connected component if and only if it finds two edge disjoint paths between them. In order to determine this, the algorithm stores in the mesh the (at most $n-1$) edges that have so far caused the merge of two connected components. These edges form a forest, and every tree in the forest represents a connected component and is called a *connectivity tree*.

When the i-th row of the adjacency matrix is read, PE_j reads the entry a_{ij}, $1 \leq j \leq n$. If $a_{ij}=1$ and $C_i \neq C_j$, the connectivity tree containing vertex i and the one containing vertex j are connected by the edge (i,j); i.e., the connected components C_i and C_j are merged. The edge (i,j) is recorded in the mesh as an edge of the newly formed connectivity tree. If $a_{ij}=1$ and $C_i=C_j$, the edge (i,j) forms a cycle in the connectivity tree representing the connected component C_i, and the algorithm (at some later stage) determines the bridge-connected components merged by the edge (i,j). If $B_i \neq B_j$, all the bridge-connected components that contain at least one vertex on the path from i (resp. j) to the lowest common ancestor of i and j in the connectivity tree (containing vertices i and j) form a new bridge-connected component. The information about the connectivity tree has to be organized such that these vertices can be determined easily. (Of course, if $B_i \neq B_j$, the edge is discarded.)

We next describe the organization of the entries of the connectivity trees. The entries representing a connected component CX are organized as edges of a rooted tree. The root of the tree is vertex CX. More precisely, every connectivity tree entry is a 6-tupel (CX, X, PX, DX, BX, DBX), where

- CX is the component number of the vertex X,
- PX is the parent node of X in the connectivity tree with root CX,
- DX is the depth of X in the connectivity tree CX,
- BX is the bridge-connected component number of X; the value of the bridge-connected component BX is always equal to the vertex in BX that has the smallest depth (i.e., is the closest to the root of the connectivity tree),
- DBX is the depth of the vertex BX.

See Figure 2.1, where the dashed undirected edges indicate edges that merged bridge-connected components. The connectivity tree entries are stored in the mesh sorted according to the component numbers CX, and entries belonging to the same connectivity tree are kept sorted according to their depth DX in the tree CX. Note that the bridge-connected component number does not correspond to the smallest vertex in this bridge-connected component, but only a minor modification is necessary to produce the output in this form.



A connectivity tree with vertices 2,4,6,7,8, and 9 in the bridge-connected component 2; the connectivity entry for vertex 9 is (1,9,7,4,2,1)
Figure 2.1

Initially, PE_i contains the connectivity tree entry $(i,i,0,0,i,0)$, but in the later stages of the algorithm there is no relation between the connectivity entry stored in PE_i and vertex i. In addition to the connected component register C_i and bridge-connected component register B_i, two other registers in PE_i are associated with vertex i throughout the algorithm:

- D_i contains the depth of vertex i in the connectivity tree with root C_i, and
- NR_i contains the number of vertices in the connectivity tree C_i.

Information about vertex i is thus kept in two different locations: in PE_i and in the connectivity tree entry for vertex i. Auxiliary registers are introduced when needed.

In the description of the implementation of our algorithm we assume that the following subroutines are available:

Random-Access-Read (RAR): PE_i requests the content of register R_j of PE_j and stores it in register R_i. This operation is denoted by $R_i := R_j$ or $R := R_j$ if the value of i is clear from the context. Note that different PE's can request data from the same PE.

SORT: specified data items in the mesh are sorted in increasing order.

PACK: k PE's in the mesh contain a 'flag', and operation PACK moves specified data stored in the flagged PE's, while maintaining their original order, into lower numbered PE (i.e., the data in the i-th flagged PE is moved into PE_i).

All of the above subroutines can be implemented to run in $O(n^{1/2})$ on a mesh of n PE's, and we refer the reader to [H1, NS2, TK] for details.

Combining the Connectivity Trees

After the PE's have read the i-th row of the matrix, the values of C_i, NR_i, and D_i stored in PE_i are broadcasted to every PE in the mesh. If there is an edge from vertex i to j, PE_j sets registers as shown in Figure 2.2.

```
for all PE_j, 1 ≤ j ≤ n pardo
        CI := C_i
        NRI := NR_i
        DI := D_i
        if a_ij = 1 then
                J := j
                CJ := C_j
                NRJ := NR_j
                DJ := D_j
odpar
```

Setting registers at the beginning of i-th iteration
Figure 2.2

The entries $(I, J, CI, CJ, NRI, NRJ, DI, DJ)$ that are created in PE's with $a_{ij}=1$ and $CI \neq CJ$ are called the ***tree-combining entries***. The algorithm next sorts the tree-combining entries in increasing order according to CJ. After the sort, the algorithm sets a flag in PE_1, and in every PE_j that contains a tree-combining entry for which the value of CJ differs from the value of CJ in PE_{j-1}. It then calls routine PACK. Assume $PE_1, \cdots, PE_p$ contain the flagged tree-combining entries (I, J, CI, CJ, NRI, NRJ, DI, DJ) after PACK. These entries represent p edges that connect $p+1$ connectivity trees, namely CI, $CJ_1, \cdots, CJ_p$. Note that throughout the description of the algorithms we refer to the value stored in a register R_i simply as R_i. The next step of the algorithm is to combine the $p+1$ connectivity trees into one. Since the connectivity tree entries are stored as edges of a rooted tree, combining connectivity trees involves 'rerooting' some of them. When a non-root vertex of a connectivity tree is made the new root, the edges on the path from the old root to the new root have to be reversed, and the depth of all the vertices in the connectivity tree has to be updated.

The rerooting of the connectivity trees is potentially a time consuming procedure, and in order to achieve the claimed time bound the algorithm never reroots the connectivity tree containing the largest number of vertices (among all the other trees to be rerooted). Thus, before the start of the rerooting process, the algorithm rearranges the tree-combining entries so that the tree-combining entry stored in PE_1 has the largest NRJ value; i.e., $NRJ_1 = \max\{NRJ_1, \cdots, NRJ_p\}$. Recall that CI, $CJ_1, \cdots, CJ_p$ are the connectivity trees to be combined, and that the registers CI, NRI, and DI of the tree-combining entries in the first p PE's have the same value, respectively.

- If $NCI_1 \geq NCJ_1$, then the connectivity tree CI containing vertex I is not rerooted. In the connectivity trees $CJ_1, \cdots, CJ_p$ vertices $J_1, \cdots, J_p$ are made the new 'roots' at depth $DI+1$. See Figure 2.3(a).
- If $NCI_1 < NCJ_1$, then the tree CJ_1 containing vertex J_1 is not rerooted. In the connectivity tree CI, vertex I is made the new root at depth DJ_1+1, and in the trees $CJ_2, \cdots CJ_p$, the vertices $J_2, \cdots, J_p$ are made the new roots at depth

DJ_1+2. See Figure 2.3(b).

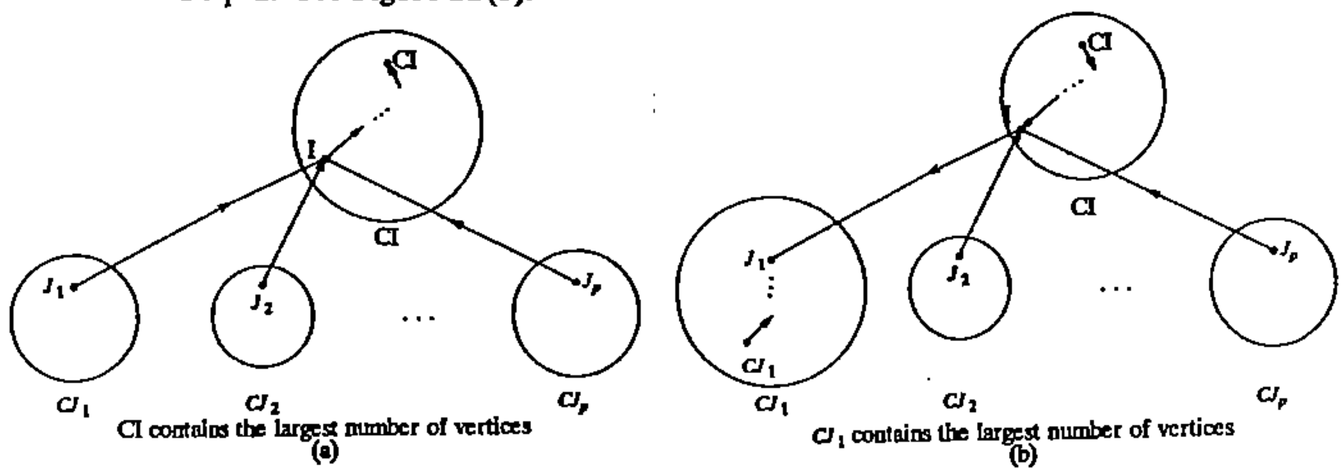


Figure 2.3

We next discuss the rerooting process for the first case (i.e., when $NCI \geq NCJ_1$) as shown in Figure 2.3(a). The second case is handled in a similar fashion. Every flagged tree-combining entry creates a *reroot entry* $(I, J, CI, CJ, ND)_r$, where ND is the new depth of vertex J which is equal to $DI+1$. Vertex J will be the new root of the vertices in the connectivity tree CJ. (Note that the subscript 'r' is used to indicate a reroot entry, not a PE.) Everyone of the p reroot entries is sent to the PE that contains the connectivity entry for vertex J; i.e., to the PE containing the connectivity entry (CX, X, PX, DX, BX, DBX) with $CX=CJ$ and $X=J$. Observe that the PE creating the reroot entry does not know the position of this connectivity entry. The position is determined by sorting all the connectivity tree entries belonging to vertices that are roots together with the p rerooting entries according to the component numbers. By doing so every reroot entry determines the position of the root of its connectivity tree in $O(n^{1/2})$ time. Once every reroot entry has been sent to the PE containing the root, it locates the connectivity entry corresponding to vertex J in $O(n^{1/2})$ time (recall that the connectivity entries of every tree CX are sorted according to their depth). Now the actual rerooting of connectivity trees CX starts, and the p connectivity trees are rerooted in parallel.

The rerooting of every tree CX works in two phases. The first phase reverses the edges on the path from vertex X to the root CX (and also updates connectivity tree entries), and the second phase updates the depth of the vertices in the subtrees rooted on a vertex on the path from X to CX. Both phases use $O(n^{1/2}+m)$ time, where m is the number of vertices in tree CX.

We now describe the implementation of *first phase* in more detail. Let (CX,X,PX,DX,BX,DBX) be a connectivity tree entry in PE_k that received the reroot entry $(I,J,CI,CJ,ND)_r$:

- If $X \neq J$, PE_k it sends the reroot entry to PE_{k-1} without changing it or its own registers.
- If $X = J$, PE_k updates its connectivity tree entry by setting $CX := CI$, $PX := I$, and $DX := ND$. PE_k then creates the update entry $(J,CI,CJ,ND)_u$ with $ND = ND+1$ and the value of registers J, CI, and CJ as in the reroot entry. The update entry remains stored in PE_k until it is activated in the second phase. PE_k next changes the reroot entry as follows. If vertex J (which, in this case, is equal to vertex X) is not the root (i.e., $X \neq CX$), PE_k then sends the reroot entry $(I,J,CI,CJ,ND)_r$ with $I = X$, $J = PX$, $ND = ND+1$, CI and CJ unchanged, to PE_{k-1}. If vertex X is the root, the second phase starts.

After the first phase, every PE containing a connectivity entry of a vertex that is incident to an edge of the tree which did get reversed, contains an update entry $(J,CI,CJ,ND)_u$. The goal of the *second phase* is to send every update entry $(J,CI,CJ,ND)_u$ to the children of vertex J (excluding the child that is now a parent), and to change the depth in the connectivity entry of the children to ND. Every child will then create its own update entry, which is to be send to its children, etc. Every PE containing a connectivity tree entry thus creates (or already contains) exactly one update entry. We now describe how to implement the second phase in $O(m)$ time. If every update entry originally in PE_k is sent (independent of the other update entries) to PE_{k+1}, PE_{k+2}, ..., and if the PE's (which contain the connectivity entries of children) create their own update entries (which are also sent to higher numbered PE's), the algorithm encounters "collisions" problems. Thus the algorithm does the following. The update entry in the root is activated first (i.e., if the connectivity entry of the root is in PE_k, PE_k sends its update entry to PE_{k+1}, PE_{k+2}, ...). Assume PE_l receives an update entry $(J, CI, CJ, ND)_u$.

- If $PX_l \neq J$ (i.e., the connectivity entry in PE_l does not belong to a child of vertex J), PE_l sends the update entry to PE_{l+1}.
- If $PX_l = J$, the algorithm sets register DX_l (of the connectivity entry) equal to ND, CX_l equal to CI, and it creates a new update entry $(J2, CI2, CJ2, ND2)$ with $J2 = X$, $CI2 = CI$, $CJ2 = CJ$, and $ND2 = ND + 1$. PE_l sends the 'old' update entry to PE_{l+1}, and keeps the newly created one until it is activated. The newly created update entry in PE_l is activated after the update entry created in PE_{l-1} passed through PE_l.

It is easy to see that this technique does not run into collision problems and that, after $O(m)$ time, where m is the number of vertices in the tree, every connectivity tree entry contains the new values.

From the above discussion it follows that the p connectivity trees can be rerooted in $O(n^{1/2}+m)$ time, where m is the number of vertices in the second largest connectivity tree involved. Before proceeding with the next major step of the algorithm, the determining and merging of bridge-connected components, we have to update the entries about vertex k in PE_k, $1 \leq k \leq n$. The number of vertices in the new connectivity tree with root CI (resp. CJ_1) can be computed in $O(n^{1/2})$ time using the p tree-combining entries. Every vertex k in $CI, CJ_1, \cdots, CJ_p$ can update its component number C_k and the value NR_k stored in PE_k in $O(n^{1/2})$ time (by using SORT twice). Finally, a write operation initiated be the connectivity entries updates the depth registers D_k in every PE_k in $O(n^{1/2})$ time.

Merging Bridge-Connected Components

After the connectivity trees have been combined, every PE_j with $a_{ij}=1$ has $C_i=C_j$, where C_i is the updated connected component number. If the edge (i,j) was used as a tree-combining edge, we set a_{ij} to 0. Next, every PE_j obtains the values B_i and D_i and, if $B_i=B_j$, also sets a_{ij} to 0, $1 \le j \le n$. The remaining PE_j's with $a_{ij}=1$ and $B_i \ne B_j$ contain an edge that merges bridge-connected components, and every such PE_j creates a *bridge entry* $(I,J,CI,BI,BJ,DI,DJ)_b$ with $I=i$ and $J=j$. The values of a bridge entries are set similar to the code shown in Figure 2.2.

While the algorithm determines the bridge-connected components merged by one bridge entry, only the section of the mesh containing the connectivity tree entries of tree CI is used. The algorithm can thus process bridge entries of different connectivity trees simultaneously. Since doing so does not affect the worst case time performance, we will not discuss this possibility in more detail. When the algorithm chooses one bridge entry $(I,J,CI,BI,BJ,DI,DJ)_b$ it follows the path from vertex I to the lowest common ancestor of I and J, referred to as lca(I,J), and the path from vertex J to lca(I,J). It marks all bridge-connected components encountered on these two paths as to be merged into one. We now describe in more detail how a bridge entry is processed in $O(bn^{1/2})$ time, where b is the number of bridge-connected components merged by the edge (I,J).

The bridge entry $(I,J,CI,BI,BJ,DI,DJ)_b$ created in PE_j is sent to the PE containing the connectivity entry of the root of connectivity tree CI. Let PE_f be this PE. At PE_f, the bridge entry is split up into two entries: $(I,CI,BI,DI)_b$ and $(J,CI,BJ,DJ)_b$, which will from now on be called the bridge entries. If $DI=DJ$, then both bridge entries are sent from PE_f to the PE containing the connectivity entry of vertex I and J, respectively. If $DI<DJ$, then only the bridge entry containing vertex J is sent, and if $DI>DJ$, then only the bridge entry containing vertex I is sent. This ensures that we move in the connectivity tree from I and J towards the lca(I,J) 'at the same pace'.

We next describe what the algorithm does once the bridge entry $(I,CI,BI,DI)_b$ has arrived at the PE containing the connectivity entry for vertex I. The action for the bridge entry for J is analogous. Assume that the connectivity tree entry for vertex I is in PE_{k1}; i.e., PE_{k1} contains the connectivity tree entry $(CX_{k1},X_{k1},PX_{k1},DX_{k1},BX_{k1},DBX_{k1})$ with $X_{k1}=I$ (and, of course, $CX_{k1}=CI$, $DX_{k1}=DI$, and $BX_{k1}=BI$). PE_{k1} sets a flag to indicate that it contains a bridge-connected component to be used in the merge.

- If $BX_{k1} = X_{k1}$, then PE_{k1} sends its bridge entry to the PE containing the connectivity entry of vertex PX_{k1}, the parent of vertex X_{k1}.
- If $BX_{k1} \neq X_{k1}$, then PE_{k1} sends its bridge entry to the PE containing the connectivity entry of vertex BX_{k1}, which is at depth DBX_{k1}. Note that by sending the bridge entry to the PE containing the entry of BX_{k1}, the algorithm never traverses edges that are in already existing bridge-connected components.

Let PE_{k2} be the PE receiving the bridge entry from PE_{k1}. The bridge entry can be sent from PE_{k1} to PE_{k2} in $O(n^{1/2})$ time. At PE_{k2}, the bridge entry $(I,CI,BI,DI)_b$ is updated to: $I=X_{k2}$, $BI=BX_{k2}$, and $DI=DX_{k2}$. The updated bridge entry is sent to PE_f. When PE_f receives the updated bridge entry (resp. entries), it checks whether the bridge-connected component containing the lca(I,J) has been reached:

- If $BI \neq BJ$, then PE_f sends out either one or both bridge entries (depending on the current depth in the bridge entries).
- If $BI = BJ$, the lowest common bridge-connected component has been reached, and PE_f sets $BNEW_f = BI$. $BNEW_f$ will be the new bridge-connected component number of all the vertices in bridge-connected components that received a flag, and the updating of bridge-connected component entries begins.

We now describe the final updating of the entries. The algorithm calls routine PACK, which places the connectivity tree entries of flagged PE's in $PE_1, \cdots, PE_s$.

Let $B_{l_1}, \cdots, B_{l_t}$ be the bridge-connected components of these entries. $BNEW_f$ is made the new bridge-connected component number of all the vertices in $B_{l_1}, \cdots, B_{l_t}$. This change has to be recorded in a number of entries: In the bridge-connected component number B_k of vertex k in PE_k, and in the bridge-connected component numbers in the connectivity entries containing vertex k. Furthermore, the entry DBX in the connectivity entries belonging to vertices of flagged bridge-connected components has to be updated. Note that the new value of DBX of all the vertices involved in the merging is the depth of vertex $BNEW_f$. The updating of all these entries can be done in $O(n^{1/2})$ time.

Theorem 2.1 The bridge-connected components can be found in time $O(n^{3/2})$ on a 2-dimensional mesh of $O(n)$ area when the graph is given in the form of an adjacency matrix.

Proof: The correctness of the algorithm follows from the preceding discussion. The time bound is obtained as follows. The time spent not on the combining of connectivity trees or the merging of bridge-connected components is $O(n^{1/2})$ for each row of the adjacency matrix. We have shown that the time used to combine and reroot connectivity trees is $O(n^{1/2}+m)$ in each iteration, where m is the number of vertices in the second largest component to be merged in the i-th iteration. In the worst case we combine and reroot connectivity trees of the same size, and we combine only 2 connectivity trees in each iteration (i.e., we combine 2 trees of $n/2$ vertices each in the n-th iteration, 2 trees of $n/4$ vertices each in the $(n-1)$-st and $(n-2)$-nd iteration, etc.) Thus, the total time spent on combining connectivity trees is

$$O((n/2 + n^{1/2}) + 2(n/4 + n^{1/2}) + 4(n/8 + n^{1/2}) + \cdots + n/2(1 + n^{1/2})),$$

which is $O(n^{3/2})$. The overall time spent on the processing of bridge entries and the merging of bridge-connected components is also $O(n^{3/2})$, since at most $n-1$ bridge-connected components can be merged. Hence, the total time of our algorithm is $O(n^{3/2})$. □

Our algorithm can be extended to find the bridge-connected components in time $O(n^{3/2})$ when the input is given in the form of edges. The overall structure of the algorithm and the entries created during the computation remain the same. Observe that now PE_k, $1 \leq k \leq n$, reads an arbitrary edge (I, J) and that the connected component number of vertex I (resp. J) is in PE_I (resp. PE_J). While the merging of bridge-connected components is done by processing the bridge entries one by one as before, the situation for combining connectivity trees is different. When the graph is given in the form of an adjacency matrix, the edges that merge connectivity trees at the i-th iteration represent a connected graph with no transitive edges. See Figure 2.3. When the graph is given in the form of edges this is no longer true. The edges between connectivity trees can now represent a graph that is not necessarily connected and that can contain transitive edges. But in order to achieve $O(n^{3/2})$ time, the connectivity trees do not have to be combined in parallel. We only have to make sure that the connectivity tree with the largest number of vertices is never rerooted. Hence, by making this step more 'sequential' the following result is obtained:

Theorem 2.2 The bridge-connected components can be found in time $O(n^{3/2})$ on a 2-dimensional mesh of $O(n)$ area when the graph is given in the form of edges.

3. Bi-Connectivity

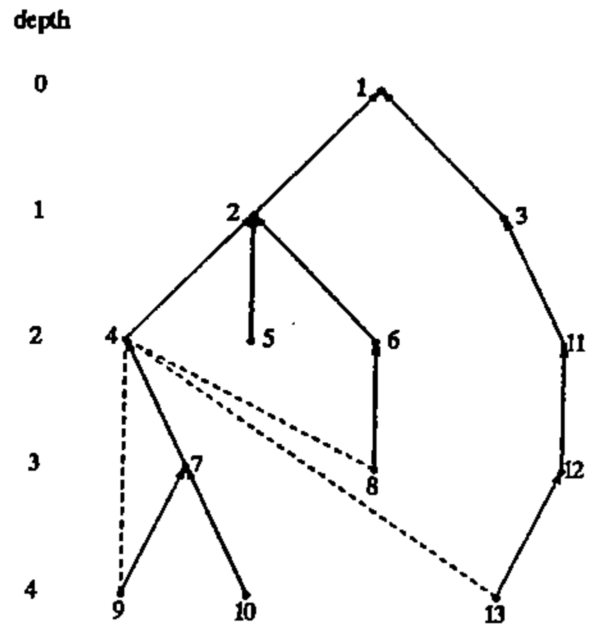
In this section we first describe an algorithm that determines the bi-connected components of an undirected graph on an $O(n)$ area mesh in time $O(n^{3/2})$ when the input is given in the form of an adjacency matrix. We also present an algorithm for input in the form of edges which runs in time $O(e+n^{3/2})$. As done for bridge-connectivity, we associate with every vertex a connected component number, and we record the edges that caused the merge of two connected components as entries of connectivity trees. The connectivity trees help to determine the bi-connected components, and the algorithm puts two vertices in the same bi-connected component if and only if it finds two vertex-disjoint paths between them. Bi-connected component numbers are used to record the bi-connectivity information obtained about the graph so far. Since one vertex can be in more than one (and at most $n/2$) bi-connected components, PE_i cannot be used to store the bi-connectivity numbers of vertex i.

The algorithm records in PE_i the entries C_i, D_i, and NR_i associated with vertex i, and they are defined as in Section 3. The bi-connectivity information is recorded in the form of *bi-number entries*, and every such entry is a 4-tupel consisting of

- a vertex,
- a bi-connected component number (the vertex is currently in),
- the vertex in the same bi-connected component number that has smallest depth in the connectivity tree, and
- the depth of this vertex.

Note that the vertex at the smallest depth in the connectivity tree cannot be used as the bi-connected component number (as done for bridge-connectivity), since this vertex could be in more than one bi-connected component. Bi-connected component numbers are now assigned as follows: PE_1 contains a register $NUMB$, which is initially set to 1. Every time a new bi-connected component is formed, it gets the number equal to the current value of $NUMB$, and $NUMB$ is increased by 1. Since every time $NUMB$ is increased, at least two bi-connected components get merged, the final value of $NUMB$ is at most $n-1$. See Figure 3.1, where the edge (4,13) is processed after the edges (4,9) and (4,8).

Every PE contains registers to store up to 2 bi-number entries; namely registers $(I1, BI1, OI1, DOI1)$ and $(I2, BI2, OI2, DOI2)$. We refer to these two sets of registers as (I^*, BI^*, OI^*, DOI^*). It is easy to show that in any graph there can be at most $(3n-3)/2$ bi-number entries, and thus two per PE are sufficient. At some time during the algorithm, the bi-number entries will be sorted according to the vertices, at other times they will be sorted according to the bi-connected component numbers. The bi-number entries are stored in packed form; i.e., the entries in PE_i are filled after the $2(i-1)$ bi-number entries in $PE_1, \cdots, PE_{i-1}$ have been filled. Initially, the



Solid arrows represent the connectivity tree,
dashed lines represent edges that merged bi-connected components;
the bi-number entries for vertex 4 are (4,1,4,2) and (4,3,1,0)
Figure 3.1

mesh contains the n bi-number entries $(i,0,i,0)$, $1 \le i \le n$.

The combining and rerooting of the connectivity trees, and the merging of connected components is done as in the bridge-connectivity algorithm. Note that a connectivity tree entry is now a 4-tupel (CX, X, PX, DX), and that after the rerooting process the DOI^* component in the bi-number entries needs to be updated.

After the combining and rerooting of the connectivity trees every PE_i with $a_{ij}=1$ and edge (i,j) not used for merging connected components creates an *edge entry* (I,J,CI,DI,DJ) with $I=i$ and $J=j$. The algorithm next finds one edge entry that forms new bi-connected components. It does so by determining in $O(n^{1/2})$ time either an edge entry that causes the merge of (at least two) bi-connected components or it concludes, also in $O(n^{1/2})$ time, that none of the (up to n) edge entries merges bi-connected components. An edge (I,J) merges bi-connected components if no bi-connected component contains both I and J. In terms of bi-number entries and edge entries this condition is stated as follows. The edge entry (I,J,CI,DI,DJ) merges bi-connected components if and only if for all bi-number entries $(I^*_k, BI^*_k,$

OI^*_k, DOI^*_k) and $(I^*_l, BI^*_l, OI_l, DOI^*_l)$ with $I^*_k = I$ and $I^*_l = J$, $BI^*_k \neq BI^*_l$ holds. It is easy to check this condition in $O(n^{1/2})$ time for one given edge entry. How one edge entry satisfying the condition is found (or it is determined that no edge entry satisfies it) in $O(n^{1/2})$ time is described next.

Selecting an Edge Entry

The algorithm adds a mark register $MARK^*_k$, $1 \leq k \leq n$, to every bi-number entry. $MARK^*_k$ is initially set to 0. The selection of an edge entry is done in three stages. In the *first stage*, the algorithm sets the mark registers in all bi-number entries of vertices adjacent to vertex I to 1; i.e., it sets $MARK^*_k = 1$ in every bi-number entry $(I^*_k, BI^*_k, OI^*_k, DOI^*_k, MARK^*_k)$ with $I^*_k = J_l$, where $(I_l, J_l, CI_l, DI_l, DJ_l)$ is an edge entry. This step is implemented in $O(n^{1/2})$ time by sorting the bi-number entries according to the vertices, then sending every edge entry $(I_l, J_l, CI_l, DI_l, DJ_l)$ to the lowest indexed PE_k containing a bi-number entry with $I^*_k = J_l$, and propagating this edge entry to higher-numbered PE's.

In the *second stage* the algorithm sets the mark registers in bi-number entries $(I^*_k, BI^*_k, OI^*_k, DOI^*_k, MARK^*_k)$ with $MARK^*_k = 1$ to 2 if there exists a bi-number entry $(I^*_l, BI^*_l, OI^*_l, DOI^*_l, MARK^*_l)$ with $I^*_l = i$ and $BI^*_k = BI^*_l$. This step is implemented in $O(n^{1/2})$ time by sorting the bi-number entries according to the bi-connected component numbers, and letting every bi-number entry with $I^*_l = i$ mark the entries with $BI^*_k = BI^*_l$.

The *third* and final stage in the selection of an edge entry the algorithm sorts the bi-number entries according to the vertices. It then selects, in $O(n^{1/2})$ time, among all edge entries (I, J, CI, DI, DJ) for which no bi-number entry corresponding to vertex J has the mark register set to 2, an arbitrary one. If one exists, the merging of bi-connected components starts. If no such edge entry is found, the i-th iteration of the algorithm is completed (and row $i+1$ of the adjacency matrix is read next).

Merging of Bi-Connected Components

After an edge entry, say (I, J, CI, DI, DJ), has been selected, the algorithm merges bi-connected components. The basic concept of the merging is similar to the one used in the algorithm for bridge-connectivity. The algorithm follows the paths from vertices I and J to the lowest common ancestor of I and J in the connectivity tree CI. Obviously, all the vertices on the two paths belong to one bi-connected component. In addition, we include a bi-connected component that contains at least two vertices that are on these two paths.

The data movement for determining the bi-connected components to be merged is similar to the one for bridge-connectivity, and we only point out some of the differences. The bi-connectivity information about a vertex is not stored in the connectivity tree entry, and it has to 'looked up' in bi-number entries. This adds an additional $O(n^{1/2})$ time for traversed every edge on the paths. Existing bi-connected components encountered on the paths are only included if they contain at least two vertices that are on the path to the lca(I, J). The algorithm uses the depth entry DOI^* of the vertex OI^* of the bi-connected component BI^* to avoid traversing more than one edge in the bi-connected component BI^*. We leave the implementation details to the reader. It follows that the time for processing one edge entry is $O(mn^{1/2})$, where m is the number of bi-connected components that get merged by the edge (I, J). Again, at most $n-1$ bi-connected components can get merged, and the overall time of the algorithm is $O(n^{3/2})$.

Theorem 3.1 The bi-connected components can be found in time $O(n^{3/2})$ on a 2-dimensional mesh of $O(n)$ area when the graph is given in the form of an adjacency matrix.

Proof: Similar to the proof of Theorem 2.1. □

We next describe how to modify the above algorithm to find the bi-connected components in $O(e+n^{3/2})$ time when the graph is given in the form of edges. Recall that for bridge-connectivity $O(n^{3/2})$ time can be achieved for both forms of input. The time-critical step in the bi-connectivity algorithm is selecting an edge entry that merges bi-connected components (or deciding that none exists) efficiently. When the idea of marking bi-number entries is applied to an arbitrary set of edges (instead of edges adjacent to vertex i), the irregularity of the input causes an increase in the time complexity. We now describe the difficulties that arise and give an informal outline how to process $O(n^{1/2})$ edge entries in $O(n)$ time.

Let (x_l, y_l) be n edges that do not merge connected components and assume they are stored in $PE_1, \cdots, PE_n$ of the mesh. Let the number of bi-number entries containing vertex x_l be be less than or equal to the number of bi-number entries containing vertex y_l. If vertex x_l is in the bi-connected components $B_l^1, \cdots, B_l^{l_l}$, form the triples (x_l, y_l, B_l^k), $1 \le k \le l_l$. Then check for every triple (x_l, y_l, B_l^k) whether or not vertex y_l is in the bi-connected component B_l^k. Unfortunately we cannot create all the triples of the n edges at once, since n edges can result in $O(n^{3/2})$ triples in the worst case, as shown by an example below.

Consider a graph with $n = 4k^2$ vertices in which vertices $x_1, \cdots, x_k$, and $y_1, \cdots, y_k$ are on one cycle, and in which every vertex x_l (resp. y_l) is in $k+1$ bi-connected components (namely the cycle and k 'triangles'). Every vertex in a triangle, except the one on the big cycle, is in exactly one bi-connected component. Let the next input sequence contain the edges (x_i, y_j), $1 \le j \le k$, $1 \le i \le k$. If we form the triples as described above, we form $(k+1)k^2 = n^{3/2}/8 + n/4$ triples. Note that no new bi-connected component is formed by these edges.

In the selection of an edge entry we handle a batch of $n^{1/2}$ edges at a time. For $n^{1/2}$ edge entries we form the triples as outlined above (note that at most $O(n)$ triples can be created), and then select an edge entry by marking bi-number entries similar to the marking step for input in the form of an adjacency matrix. Once an edge entry has been selected and bi-connected components been merged, the next edge entry is selected from the current batch of $n^{1/2}$ edges in $O(n^{1/2})$ time. Thus the total time for processing n edge entries (not counting the time to merge bi-connected components) in $O(n)$. The overall time spent in selecting edge entries is $O(\frac{e}{n} n) = O(e)$. The time spent in the other steps of the algorithm remains the same. We can thus state the following theorem:

Theorem 3.2 The bi-connected components can be found in time $O(e + n^{3/2})$ on a 2-dimensional mesh of $O(n)$ area when the graph is given in the form of edges.

References

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

[AK] M.J. Atallah, S.R. Kosaraju, 'Graph Problems on a Mesh-Connected Processor Array', *Journal of ACM*, Vol. 31, pp 649-667, 1984.

[DNS] E. Dekel, D. Nassimi, S. Sahni, 'Parallel Matrix and Graph Algorithms', *SIAM Journal on Computing*, Vol. 10, pp 657-675, 1981.

[GKT] L.J. Guibas, H.T. Kung, C.D. Thompson, 'Direct VLSI Implementations for Combinatorial Algorithms', *Proc. of Conf. VLSI tech. design and fabrication*, Caltech 1979.

[H1] S.E. Hambrusch, 'The Complexity of Graph Problems on VLSI', Ph.D. Thesis, The Pennsylvania State University, August 1982.

[H2] S.E. Hambrusch, 'VLSI Algorithms for the Connected Component Problem', *SIAM Journal on Computing*, Vol. 12, pp 354-365, 1983.

[HCS] D.S. Hirschberg, A.K. Chandra, D.V. Sarwate, 'Computing Connected Components on Parallel Computers', *CACM*, pp 461-464, Aug. 1979.

[HMS] P.H. Hochschild, E.W. Mayr, A.R. Siegel, 'Techniques for Solving Graph Problems in Parallel Environments', *Proc. of 24-th IEEE FOCS Conf.*, pp 351-359, 1983.

[JS] J. Ja'Ja', J. Simon, 'Parallel Algorithms in Graph Theory: Planarity Testing', *SIAM Journal on Computing*, pp 314-328, May 1982.

[KL] H.T. Kung, C.E. Leiserson, 'Systolic Arrays for VLSI', appeared in *Introduction to VLSI Systems*, C. Mead, L. Conway, Addison-Wesley, pp 260-292, 1980.

[LV] R.J. Lipton, J. Valdes, 'Census Function: An Approach to VLSI Upper Bounds', *Proc. of the 22-nd Ann. Symp. on Found. of Comp. Sc.*, pp 13-22, 1981.

[MS1] R. Miller, Q.F. Stout, 'Computational Geometry on a Mesh-Connected Computer", *Proc. of 1984 Internat. Conf. on Parallel Processing*, pp 66-73, 1984.

[MS2] R. Miller, Q.F. Stout, 'Geometric Algorithms for Digitizes Pictures on a Mesh-Connected Computer', to appear in *IEEE Tran. on Pattern Analysis and*

Machine Intelligence.

[NS1] D. Nassimi, S. Sahni, 'Finding Connected Components and Connected ones on a Mesh-connected Parallel Computer', *SIAM J. on Comp.*, pp 744-757, 1980.

[NS2] D. Nassimi, S. Sahni, 'Data Broadcasting in SIMD Computers', *IEEE Transactions on Computers*, pp 101-106, 1981.

[SJ] C. Savage, J. Ja'Ja', 'Fast, Efficient Parallel Algorithms for some Graph Problems', *SIAM J. on Comp.*, pp 682-691, 1981.

[SV] Y. Shiloach, U. Vishkin, 'An $O(\log n)$ parallel connectivity algorithm', *J. of Algorithms*, Vol. 3, pp 57-67, 1982.

[TK] C. Thompson, H. Kung, 'Sorting on a Mesh-Connected Parallel Computer', *CACM*, pp 263-271, 1977.

[TC] Y.H. Tsin, F.Y. Chin, 'Efficient Parallel Graph Algorithms for a Class of Graph Theoretic Problems', *SIAM J. on Computing*, pp 580-599, 1984.

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