Parallel algorithms for bridge- and bi-connectivity on minimum area meshes

by SUSANNE E. HAMBRUSCH
Purdue University
West Lafayette, Indiana

ABSTRACT

I 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 an $n^{1/2} \times n^{1/2}$ mesh of processing elements. The algorithms find the bridge-connected components in $O(n^{3/2})$ time for input both in the form of an adjacency matrix and in the form of edges. For bi-connectivity, I show how to achieve $O(n^{3/2})$ time when the input is in adjacency matrix form, and $O(e + n^{3/2})$ time when the input is in the form of edges.
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. Numerous researchers have developed parallel algorithms tailored towards the mesh. In this paper, I 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 \( n \) processing elements (PEs). It is assumed that every input is read only once, every output is generated once, and every PE contains a constant number of registers of \( \log n \) bits each. Under these assumptions, every algorithm finding the bridge-, bi-, and connected components requires at least \( n \) PEs. Hence, my algorithms run on a minimum area network. Developing algorithms for minimum area networks is of theoretical interest because of the algorithm technique developed, and of practical interest because area is an expensive resource.

The \( n^{1/2} \times n^{1/2} \) mesh receives the input representing \( G \) in a number of input waves. It cannot store the inputs of all the input waves for the entire computation. Observe that numerous problems (e.g., directed graph and sorting) cannot be solved on networks with storage capacity less than the length of their input. I 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; in the case of edges, the \( i \)th input wave consists of \( n \) arbitrary edges of \( G \). In the \( i \)th input wave \( PE_j \), \( 1 \leq j \leq n \), receives exactly one input (which is either the bit \( a_{ij} \) or an edge \((x_j, y_j)\)). I assume that the input waves are received in a when- indeterminate mode (i.e., the time at which the \( i \)th input wave is read may depend on the data). I show how to determine the bridge-connected components in \( O(n^{3/2}) \) time for both adjacency matrix and edge input. For bi-connectivity, I show how to achieve \( O(n^{3/2}) \) time when the input is in adjacency matrix form, and \( O(e + n^{3/2}) \) time when the input is in the form of edges.

Algorithms for graph problems on parallel models with enough PEs and memory to store a representation of the graph explicitly during the entire computation have been studied extensively for a variety of parallel models. The issues involved when only part of the input is available at any time during the algorithm, and where this input has to be processed (i.e., irrelevant inputs are discarded) before the next input wave can be read, are quite different. Lipton and Valdes and Hochschild et al. consider binary tree networks with \( n \) leaves for solving graph problems with adjacency matrix input. The algorithms in Hochschild et al. require \( \log n \) registers per PE, and the bi-connectivity algorithm in Lipton and Valdes has to read the adjacency matrix twice. I have previously used the model in this paper and described algorithms on \( O(n) \) area meshes for finding the connected components for both forms of input.

In my algorithms, I assume that the \( n \) PEs, \( PE_1, \ldots, PE_n \), 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 the time bounds hold when other standard indexing schemas are used. The time bounds of the algorithms are further independent of how the \( i \)th input wave is being input. If only \( n^{1/2} \) PEs on the boundary of the mesh can perform the I/O, the additional \( n^{1/2} \) time needed to read the \( i \)th input wave does not change the time bounds.

The algorithms for bridge-connectivity associate with every vertex of the graph a connected and a bridge-connected component number. The output consists of a bridge-connected component number for every vertex. This number does not correspond to the index of the smallest vertex in this bridge-connected component (how it is determined is described in the section Bridge Connectivity), but our algorithms can easily be modified to produce output of this form. The algorithms for bi-connectivity also number the bi-connected components. The output lists for every vertex the bi-connected components containing this vertex. Observe that bi-connectivity induces an equivalence relation on the edges, while bridge-connectivity induces one on the vertices.

My algorithms process the \( i \)th input wave completely before reading the \((i + 1)\)th input wave. Processing an input wave consists of determining which inputs are irrelevant (and discarding them) and incorporating the relevant input (i.e., the input which contains new information about the bridge- (resp. bi-) connected components) into the data structures used on the mesh. The elements of the data structures are distributed among PEs in a way that allows for fast data movement and thus fast processing of an input wave.

BRIDGE-CONNECTIVITY

In this section, I present an algorithm for finding the bridge-connected components on a two-dimensional mesh of \( O(n) \) area in time \( O(n^{3/2}) \). I 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.

Let us start with an informal description of the approach used in the bridge-connectivity algorithm. Every 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, \( 1 \leq i \leq n \). These two entries are stored in \( PE_i \) in the mesh. Initially, \( B_i = C_i = i, 1 \leq i \leq n \). The algorithm puts two vertices in the same bridge-connected component iff there exist two edge-disjoint paths between them. In order to determine this, the algorithm stores in the mesh the edges (at most \( n - 1 \)) that have so far caused the merge of two connected components. These edges form a forest where every tree represents a connected component and is called a connectivity tree.

When the \( i \)th row of the adjacency matrix is read, \( PE_i \) 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 through 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 determines the bridge-connected components merged by the edge \((i, j)\). If \( B_i = B_j \), 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.

I next describe the organization of the entries of the connectivity trees. The entries representing the connected component \( CX \) have the form of edges of a rooted tree. The root of the tree is vertex \( CX \). More precisely, every connectivity tree entry is a six-tupel \((CX, X, PX, DX, BX, DBX)\) where

\[
\begin{align*}
CX & \text{ is the component number of the vertex } X, \\
PX & \text{ is the parent node of } X \text{ in the connectivity tree with root } CX, \\
DX & \text{ is the depth of } X \text{ in the connectivity tree } CX, \\
BX & \text{ is the bridge-connected component number of } X; BX \text{ is always equal to the vertex in } BX \text{ that has the smallest depth (i.e., is the closest to the root of the connectivity tree),} \\
DBX & \text{ is the depth of the vertex } BX.
\end{align*}
\]

In Figure 1, the broken undirected edges indicate edges that merged bridge-connected components. Connectivity tree entries are stored in the mesh, sorted according to the component numbers \( CX \). Entries belonging to the same connectivity tree are kept sorted according to their depth \( DX \) in the tree.

Initially, \( PE_i \) contains the connectivity tree entry \((i, i, 0, 0, 0, 0)\), 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 \( i \), \( D_i \), which contains the depth of vertex \( i \) in the connectivity tree with root \( C_i \), and \( NRI_i \), which contains the number of vertices in the connectivity tree \( C_i \). Information about vertex \( i \) is thus kept both 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 the algorithm, I assume that the following subroutines are available:

\[
\text{Random-Access-Read (RAR): } PE_i \text{ requests the content of register } R_i \text{ of } PE_i \text{ and stores it in register } R_i. \text{ This operation is denoted by } R_i = R_i \text{ or } R_i = R_i \text{ if the value of } i \text{ is clear from the context. Note that different PEs can request data from the same PE.} \\
\text{SORT: Specified data items in the mesh are sorted in increasing order.} \\
\text{PACK: } k \text{ PEs in the mesh are "flagged." PACK moves specified data in the flagged PEs (while maintaining their original order) into lower numbered PEs (i.e., the data in the } j \text{th flagged PE is moved into } PE_j. \\
\text{All of the above subroutines can be implemented to run in } O(n^{1/2}) \text{ on a mesh of } n \text{ PEs.}^{3,7,19}
\]

**Combining the Connectivity Trees**

After the PEs have read the \( i \)th row of the matrix, the values of \( C_i, NR_i, \) and \( D_i \) stored in \( PE_i \) are distributed 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.

The entries \((i, j, CI, CJ, NRI, NRI, DI, DJ)\) that are in PEs with \( a_{ij} = 1 \) and \( CI \neq CJ \) are called the tree-combining entries. The algorithm sorts the tree-combining entries in increasing order according to \( CI \). After the sort, the algorithm sets a flag in \( PE_1 \), and in every \( PE_i \) that contains a tree-combining entry for which the value of \( CI \) differs from the value of \( CJ \) in \( PE_{i-1} \). It then calls routine PACK. Assume \( PE_1, \ldots, PE_p \) contain the flagged tree-combining entries.
for all \( PE_i, 1 \leq j \leq n \) pardo
\[ X \]
\[ J = j \]
\[ C_j := C_j \]
\[ NR_j := NR_j \]
\[ D_j := D_j \]
\[ if \ a_p = 1 \ then \ J = j \]
\[ C_j := C_j \]
\[ NR_j := NR_j \]
\[ D_j := D_j \]
\[ odpar \]

Figure 2—Setting registers at the beginning of the \( i \)th iteration

(I, J, CI, CJ, NRI, NRJ, DI, DJ) after PACK. These entries represent \( p \) edges that connect \( p + 1 \) connectivity trees, namely CI, CJ, ... , Ck. Note that, throughout the description of the algorithms, I refer to the value stored in a register \( R \) simply as \( R \). 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, we have to "reroot" some connectivity trees. When a non-root vertex of a connectivity tree becomes the new root, the edges on the path from the old root to the new have to be reversed, and the depth of all vertices in the connectivity tree has to be updated.

The rerooting of the connectivity trees is a potentially time consuming procedure. 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 entry stored in \( PE_i \) has the largest \( NRi \) value (i.e., \( NRi = \max \{ NRj \} \)). Recall that CI, CJ, ... , CK are the connectivity trees to be combined, and that the entries CI, CJ, and DI have the same value in the \( p \) PEs.

1. If \( NCI > NCI_j \), then the connectivity tree CI containing vertex \( J \) is not rerooted. In the connectivity trees \( Cj, ... , Ck \), vertices \( J, ... , J \) are the new roots at a depth of \( DI + 1 \) (see Figure 3(a)).

2. If \( NCI < NCI_j \), then the tree CI containing vertex \( J \) is not rerooted. In the connectivity tree CI, vertex \( J \) is made the new root at depth \( DI + 1 \), and in the trees \( Cj, ... , Ck \), the vertices \( J, ... , J \) are made the new roots at a depth of \( DI + 2 \) (see Figure 3(b)).

I next discuss the rerooting process for the first case (i.e., \( NCI > NCI_j \)). The second case is handled in a similar fashion. Every tree-combining entry creates a root entry \((I, J, CI, CJ, ND)\), where \( ND \) is the new depth of vertex \( J \) in the \( p \) roots for the \( p \) root entries is sent to the PE that contains the connectivity entry for vertex \( J \) (i.e., to the PE containing the entry \((CX, X, PX, DX, BX, DBX) \) with \( CX = CI \) and \( X = J \)). Observe that the PE creating the root 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, and the \( p \) rerooting entries according to the component numbers. By doing so, every root entry determines the position of the root of its connectivity tree in \( O(n^{1/2}) \) time. Once every root 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 \( CX \).

I now describe the implementation of the 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 \((I, J, CI, CJ, ND)\).

1. If \( X = J \), \( PE_k \) sends the root entry to \( PE_{k-1} \) without changing it or its own registers.

2. If \( X = J \), \( PE_k \) updates its connectivity tree entry by setting \( CX := CI \), \( PX := I \), and \( DX := ND \). Then, \( PE_k \) creates the update entry \((I, CI, CJ, ND)\), with \( ND = ND + 1 \) and the value of registers \( I, CI, and CJ \) as in the root entry. The update entry remains stored in \( PE_k \) until it is activated in the second phase. Next, \( PE_k \)
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 ≠ CX), PE sends the reroot entry (I, J, CI, CI, ND), with I = X, J = PX, ND = ND + 1, CI and CJ unchanged, to PE. 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 got reversed contains an update entry (I, CI, CI, ND). The goal of the second phase is to send every update entry (J, CI, CI, ND), 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 to be sent to its children, etc. Every PE containing a connectivity tree entry thus creates (or already contains) exactly one update entry.

I now describe how to implement the second phase in $O(n^{1/2} + m)$ time. Every update entry originally in PE is sent (independent of the other update entries) to PE, PE, PE, ..., and if the PEs (which contain the connectivity entries of children) create their own update entries (which are also sent to higher numbered PEs), the algorithm encounters congestion problems. This, 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, PE sends its update entry to PE, PE, PE, ...). Assume PE receives an update entry (I, CI, CI, ND),

1. If $PX ≠ J$ (i.e., the connectivity entry in PE does not belong to a child of vertex J), PE sends the update entry to PE,.
2. If $PX = J$, the algorithm sets register $DX$ (of the connectivity entry) equal to ND, CX, equal to CI, and it creates a new update entry $(I', Ci', Ci', ND')$ with $I' = X, Ci' = Cl, Ci' = CI, and ND' = ND + 1$. PE sends the old update entry to PE, and keeps the newly created one until it is activated. The newly created update entry in PE is activated after the update entry created in PE, passes through PE,.

It is easy to see that this technique does not run into congestion 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, 1 ≤ k ≤ n. The number of vertices in the new connectivity tree with root CI (resp. CI) can be computed in $O(n^{1/2})$ time using the p tree-combining entries. Every vertex k in CI, CI, ..., CI can update its component number Cl and the value NR to the new values in $O(n^{1/2})$ time (by using SORT twice). A write operation initiated by the connectivity entries updates the depth registers $D_{k}$ in every PE in $O(n^{1/2})$ time.

Merging Bridge-Connected Components

After the connectivity trees have been combined, every PE, with $a_{0} = 1$ has $C_{i} = C$, where $C$ is the updated connected component number. If the edge $(i, j)$ were used as a tree-combining edge, we set $a_{0}$ to 0. Next, every PE obtains the values $Bi$ and $Di$, and if $Bi = Bi$ also sets $a_{0}$ to 0, 1 ≤ j ≤ n. Every remaining PE with $a_{0} = 1$ and $Bi ≠ Bi$ contains an edge that merges bridge-connected components and creates the bridge entry (I, J, CI, BI, BJ, DI, DJ). The values of a bridge entry are set similar to the code shown in Figure 2.

While the algorithm determines the bridge-connected components merged by a 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, I will not discuss this possibility in more detail. When the algorithm chooses one bridge entry (I, J, CI, BI, BJ, DI, DJ), 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. I 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) is sent to the PE containing the connectivity entry of the root of connectivity tree CI. Let $PE_{i}$ be this PE. At $PE_{i}$, the bridge entry is split up into two entries, (I, CI, BI, DI) and (J, CI, BJ, DJ), which will from now on be called the bridge entries. If $DI = DJ$, then both bridge entries are sent from $PE_{i}$ 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 J is sent. This ensures that we move in the connectivity tree from I towards the lca(I, J) "at the same pace."

I next describe what the bridge entry (I, CI, BI, DI) does. The action for the bridge entry for J is analogous to it. Assume that the connectivity tree entry for vertex I is in $PE_{i}$ (i.e., $PE_{i}$ contains entry $(CX_{i}, XI_{i}, PX_{i}, DX_{i}, BX_{i}, DBX_{i})$ with $XI_{i} = I$ and $CX_{i} = CI, DX_{i} = DI, and BX_{i} = BI$). $PE_{i}$ sets a flag to indicate that it contains a bridge-connected component to be used in the merge.

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

Let $PE_{i}$ be the PE receiving the bridge entry from $PE_{i}$. The bridge entry can be sent from $PE_{i}$ to $PE_{i}$ in $O(n^{1/2})$ time. At
iteration. In the worst case we combine and reroot connectivity trees of the same size, and we combine only two connectivity trees in each iteration (i.e., we combine two trees of the same number that received a flag, and the updating of bridge-connected component entries begins.

I now describe the final updating of the entries. The algorithm calls routine PACK, which places the flagged connectivity entries containing bridge-connected components to be merged in $PE_1, \ldots, PE_i$. Let $B_1, \ldots, B_n$ be these bridge-connected components. $BMIN_i$ is 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.

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

BI-CONNECTIVITY

In this section, I 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. As in the algorithm for bridge-connectivity, associate with every vertex a connected component number, and record the edges that caused the merge of two connected components as entries of connectivity trees. 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 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 there are two vertex-disjoint paths between them. The algorithm records in $PE_i$ the entries $C_i$, $D_i$, and $NR_i$ associated with vertex $i$ as defined in the previous algorithm. It records the bi-connectivity information in the form of bi-number entries. Every such entry is a four-tupel consisting of

1. A vertex,
2. A bi-connected component number (that the vertex is currently in),
3. The vertex in the same bi-connected component number that has smallest depth in the connectivity tree, and
4. The depth of this vertex.

The vertex at the smallest depth in the connectivity tree cannot be used as the bi-connected component number, because this vertex could be in more than one bi-connected component. Bi-connected component numbers are now assigned as
follows. PE_i 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. Because every time NUMB is increased, at least two bi-connected components get merged, the final value of NUMB is at most \( n - 1 \).

Every PE contains registers to store up to two bi-number entries, namely registers \((I_1, B_{I_1}, O_{I_1}, DO_{I_1})\) and \((I_2, B_{I_2}, O_{I_2}, DO_{I_2})\). We refer to these two sets of registers as \((P, B_i, O_i, D_{Oi})\). It is easy to show that in any graph there can be at most \((3n - 3)/2\) bi-number entries. Thus, two per PE are sufficient. At some time during the algorithm, the bi-number entries are sorted according to the vertices; at other times, the entries are 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 \(PE_1, \ldots, PE_{i-1}\) have been filled). Initially, the mesh contains the \(n\) bi-number entries \((i, 0, i, 0), 1 \leq i \leq n\).

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 four-tupel \((CX, X, PX, DX)\), and that after the rerooting process, the \(DO_i\) component in the bi-number entries needs to be updated.

After the combining and rerooting of the connectivity trees, every PE_i with \(a_{ik} = 1\) and edge \((i, j)\) not used in the connected component merging process contains an edge entry \((I, J, CI, DI, DJ)\) with \(I = i\) and \(J = j\). Next, the algorithm finds one edge entry that forms new bi-connected components. It determines in \(O(n^{3/2})\) time either an edge entry that causes the merge of at least two bi-connected components, or it concludes in \(O(n^{3/2})\) 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 for all bi-number entries \((I_2, B_{I_2}, O_{I_2}, D_{O_{I_2}})\) and \((I_2, B_{I_2}, O_{I_2}, D_{O_{I_2}})\) with \(I = i\) and \(I = j\), \(B_{I_2} = B_{I_2}\) holds. It is easy to check this condition in \(O(n^{3/2})\) time for any given edge entry. How one edge entry satisfying the condition is found, or how it is determined that no edge entry satisfies it in \(O(n^{3/2})\) time is described next.

**Selecting an Edge Entry**

The algorithm adds a mark register \(MARK_i^k\), \(1 \leq k \leq n\), to every bi-number entry. \(MARK_i^k\) 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 \(MARK_i^k = 1\) in every bi-number entry \((I, B_{I_2}, O_{I_2}, D_{O_{I_2}}, MARK_i^k)\) with \(I = I\), where \((I, J, CI, DI, DJ)\) is an edge entry). This step is implemented in \(O(n^{3/2})\) time by sorting the bi-number entries according to the vertices, then sending every edge entry \((I, J, CI, DI, DJ)\) to the lowest indexed PE_i containing a bi-number entry with \(I = I\), and propagating this edge entry to higher-numbered PEs.

In the second stage, the algorithm sets the mark registers in bi-number entries \((I, B_{I_2}, O_{I_2}, D_{O_{I_2}}, MARK_i^k)\) with \(MARK_i^k = 1\) to 2 if there exists a bi-number entry \((I, B_{I_2}, O_{I_2}, D_{O_{I_2}}, MARK_i^k)\) with \(I = i\) and \(B_{I_2} = B_{I_2}\). This step is implemented in \(O(n^{3/2})\) time by sorting the bi-number entries according to the bi-connected component numbers, and letting every bi-number entry with \(I = i\) mark the entries with \(B_{I_2} = B_{I_2}\).

In the third stage 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 \(O(n^{3/2})\) 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 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 \(Cl\). Obviously, all the vertices on the two paths belong to one bi-connected component. In addition, 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; I only point out some of the differences. The bi-connectivity information about a vertex is not stored in the connectivity tree entry; it has to be “looked up” in bi-number entries. This adds an additional \(O(n^{3/2})\) time for traversing 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 \(DO_i^k\) of the vertex \(O_i^k\) of the bi-connected component \(BI_i^k\) to avoid traversing more than one edge in the bi-connected component. I leave the implementation details to the reader. It follows that the time for processing one edge entry is \(O(mn^{3/2})\), 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:** The bi-connected components can be found in time \(O(n^{3/2})\) on a two-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 1. \(\square\)

I 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 input in the form of edges. 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 arbi-
trary set of edges instead of edges adjacent to vertex i, the 
irregularity of the input causes an increase in the time 
complexity.

I now describe the difficulties that arise and give an infor-
mal outline how to process $O(n^{1/2})$ edge entries in $O(n)$ time. 
Let $(x, y)$ be $n$ edges that do not merge connected com-
ponents and assume they are stored in $PE_1, \ldots, PE_n$ of 
the mesh. Let the number of bi-number entries containing vertex 
x be less than or equal to the number of bi-number entries 
containing vertex $y$. If vertex $x_i$ is in the bi-connected com-
ponents $B_i^1, \ldots, B_i^5$, form the triples $(x_i, y_i, B_i^5)$, $1 \leq k \leq l$. Then check for every triple $(x, y, B_i^5)$ whether or not vertex 
y is in the bi-connected component $B_i^5$. Unfortunately, we 
cannot create all the triples of the $n$ edges at once, because $n$ 
edges can result in $O(n^{12})$ triples in the worst case.

Consider a graph in which every one of the vertices 
$x_1, \ldots, x_{n^{1/2}}$ is currently in $n^{1/2}$ bi-connected components and 
every one of the vertices $y_1, \ldots, y_{n^{1/2}}$ is currently in $n^{1/2}$ bi-
connected components. Let the next input sequence contain 
the edges $(x_i, y_j)$, $1 \leq j \leq n^{1/2}$, $1 \leq i \leq n^{1/2}$. Then every edge 
$(x_i, y_j)$ creates $n^{12}$ triples, and altogether $n^{12}$ triples are 
created. This bound is achieved by a graph that has $n = 4k^2$ 
vertices in which vertices $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$ are on 
one cycle, and in which every vertex $x_i$ (resp. $y_i$) is in $k$ 
bi-connected components (namely the cycle and $k$ "triangles." The edges $(x, y)$ form $k^2 = n^{2/4} \sqrt{2}$ triples, but no new bi-connected components.

In the selection of an edge entry, we handle a batch of $n^{12}$ 
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 merged, the next edge entry is se-
lected from the current batch of $n^{12}$ edges in $O(n^{12})$ time. Thus, the total time for processing $n$ edge entries (not counting the time to merge bi-connected components) is $O(n)$. The overall time spent in selecting edge entries is 

$$O\left(\frac{e}{n}\right) = O(e).$$

The time spent in the other steps of the algorithm remains the same. We can thus state the following.

**Theorem 4:** The bi-connected components can be found in time $O(e + n^{12})$ on a two-dimensional mesh of $O(n)$ area when the graph is given in the form of edges.

**ACKNOWLEDGMENT**

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

**REFERENCES**

1. Atallah, M. J., and S. R. Kosaraju. “Graph Problems on a Mesh-
tation for Combinatorial Algorithms.” Proceedings of the Caltech Confer-
ence on VLSI Technical Design and Fabrication. Pasadena, Calif.: California 
Institute of Technology, 1979.
3. Hambrusch, S. E. “The Complexity of Graph Problems on VLSI.” Ph.D. 
and L. Conway (eds.), Introduction to VLSI Systems. Reading, Mass.: 
Addison-Wesley, 1980.
5. Miller, R., and Q. F. Stout. “Computational Geometry on a Mesh-
Connected Computer.” IEEE Transactions on Pattern Analysis 
and Machine Intelligence, 10.1109/PAMI.1988.4766727.
on a Mesh-Connected Computer.” IEEE Transactions on Pattern Analysis 
7. Thompson, C., and H. Kung. “Sorting on a Mesh-Connected Parallel 
8. Ullman, J. D. Computational Aspects of VLSI. Rockville, Md.: 
9. Dekel, E., D. Nassimi, and S. Sahni. Parallel Matrix and Graph Algo-
nected Components on Parallel Computers.” Communications of the ACM, 
nected Ones on a Mesh-Connected Parallel Computer.” SIAM Journal on 
Class of Graph Theoretic Problems.” SIAM Journal on Computing, 13 
16. Lipton, R. J., and J. Valdes. “Census Function: An Approach to VLSI 
Upper Bounds.” Proceedings of the 22nd Annual Symposium on Founda-
pp. 13-22.
Graph Problems in Parallel Environments." Proceedings of 24th IEEE 
pp. 351-359.
18. Hambrusch, S. E. “VLSI Algorithms for the Connected Component Prob-