International Journal of Parallel and Distributed Systems and Networks, Vol. 3, No. 4, 2000

FINDING THE MOST VITAL EDGE FOR GRAPH MINIMIZATION PROBLEMS ON MESHES AND HYPERCUBES

Weifa Liang,* Xiaojun Shen,** and Qing Hu***

Abstract

Let $G(V, E, w)$ be an undirected, weighted, connected simple graph. Let $\mathcal{P}$ be a minimization problem in G. Edge $e^* \in E$ is called the most vital edge if its removal from G maximizes the value of $\mathcal{P}$ in $G(V, E - \{e^*\}, w)$. This paper considers the most vital edge with respect to the minimum spanning tree problem and the single-source shortest path problem. An $O(n)$ optimal algorithm for finding the most vital edge with respect to these two problems is presented on an $n \times n$ mesh. The algorithm is simulated on an $n \times n \times n$ hypercube array, and an $O(\log^2 n)$ time algorithm is then obtained, which is faster than a previously known result on the same model.

Key Words

Minimum spanning tree, mesh-connected processor array, hypercube-connected processor array, shortest path, network optimization, most vital edge, parallel algorithms

1. Introduction

Let $G(V, E, w)$ be an undirected, weighted graph, where $|V| = n$. Let $\mathcal{P}$ be a minimization problem. The most vital edge with respect to (w.r.t.) $\mathcal{P}$ is an edge defined $e^* \in E$ such that its removal from G maximizes the value of $\mathcal{P}$ in the remaining graph $G' = G(V, E - \{e^*\}, w)$. The most vital edge problem w.r.t. minimum spanning trees (MSTs) has been extensively studied in the past few years. For example, Hsu *et al.* [1] and Iwano *et al.* [2] presented efficient sequential algorithms for it. Hsu *et al.* [3], Shen [4], and Liang *et al.* [5] proposed NC algorithms for this problem on a various PRAMs. Hsu *et al.* [3] also presented an $O(n^2)$ work parallel algorithm on a hypercube model. Their algorithm on this model needs $O(n^{1+x})$ time using $O(n^{1-x})$ processors, $0 < x < 1$. An extension of this problem is the k most vital edge problem w.r.t. MSTs with $k > 1$ [6, 7]. Some results for this extension have been reported [5–9].

For the most vital edge problem w.r.t. single-source shortest path (SSP), a number of researchers have made some contributions in the past decade. Corley *et al.* [10] first raised this problem and gave some preliminary results. Malik *et al.* [11] later presented an efficient algorithm for undirected graphs. For the k most vital edge problem w.r.t. SSP, Corley *et al.* [10] stated that the identification of the sufficient condition leading to a constructive procedure to solve the k most vital edge problem is an open problem. Malik *et al.* [11] claimed that they found this sufficient condition by presenting an exponential algorithm for the problem. Unfortunately, their algorithm is incorrect, as was found by Bar-Noy *et al.* [12], who gave a counterexample. Ball *et al.* [13] generalized the k most vital edge problem by assigning removal cost to every edge, and the total removal cost is bounded. For this generalized problem, they have shown that it is NP-complete. However, their proof does not imply that the k most vital edge problem is NP-complete too. Bar-Noy *et al.* [12] have shown that the k most vital edge problem is also NP-complete. Recently, Liang *et al.* [14] gave a fast polynomial algorithm for this problem when k is fixed, and Venema *et al.* [15] gave an NC algorithm for the case $k = 1$.

In this paper we will consider the most vital edge problem w.r.t. MST and SSP on the mesh and hypercube arrays. We first present an $O(n)$ optimal time algorithm on an $n \times n$ mesh array for the two problems. It is time-optimal because a trivial lower bound for global communication on this model needs $\Omega(n)$ time. We then simulate this algorithm on an $n \times n \times n$ hypercube array. Thus, an $O(\log^2 n)$ time algorithm is obtained, which is faster than a previously known result [3]. It must be pointed out that Hsu *et al.* [3] once presented an $O(n^2)$ work parallel algorithm for the most vital edge problem w.r.t. MSTs on a hypercube model too. Although we both use the hypercube as our computational model, there is some difference in the definition of the model. They allow each processor to have size of $f(n)$ local memory where $f(n)$ is a function of n, whereas we only allow each processor to have a constant number of registers (constant memory). As a result, their

* Department of Computer Science, Australian National University, Canberra, ACT 0200, Australia; e-mail: wliang@cs.anu.edu.au

** Computer Science Telecommunications Program, University of Missouri-Kansas City, 5100 Rockhill Road, Kansas City, MO 64110; e-mail: xshen@cstp.umkc.edu

*** FutureNet Technologies Corporation, 2610 S. California Ave., Suite F, Monrovia, CA 91016 USA

(paper no. 204-0138)

algorithm needs $O(n^{1+x}) \geq \Omega(n)$ time but our algorithm for the same problem needs only $O(\log^2 n)$ time. It is not difficult to see that our better time bound is achieved by making use of more processors.

Without loss of generality, we assume that $G(V, E, w)$ is two-edge connected. Otherwise, when we delete an edge that is a bridge of G, the remaining graph is disconnected, and hence no MST or any shortest path in it can exist.

The rest of this paper is organized as follows. In Section 2 we define the computational models. The algorithms for the most vital edge problem w.r.t. both MST and SSP on an $n \times n$ mesh array are presented in Section 3, and the algorithm on an $n \times n \times n$ hypercube array is given in Section 4. Section 5 concludes the paper.

2. The Computational Models

2.1 The 2-D Mesh Array

In this paper we use the model described by Atallah *et al.* [16]. The computational model consists of $N = n \times n$ identical processors positioned on a square array, and the processor positioned at (i, j) is represented by $PE_{i,j}$ whose index is $in+j$, where $0 \leq i, j \leq n-1$, that is, the processors are arranged in n rows and n columns. Each processor can communicate with its four neighbours provided they exist. A processor has a constant number of registers, each of which can store a word of $c \log n$ bits for some constant $c > 0$. A processor can proceed with the usual arithmetic and Boolean operations in one unit of time. There is a single instruction stream: within one time unit, the same instruction is broadcast to all processors and executed simultaneously. A single instruction at a processor consists of an arithmetic operation, transmitting a word to all neighbours and receiving a word from its neighbours. We refer to this model as a *2-D mesh array*.

2.2 The Hypercube Array

A q-dimensional hypercube array consists of 2^q processors indexed from 0 to $2^q - 1$. Let $i_{q-1}i_{q-2}\dots i_0$ be the binary representation of i, $0 \leq i < p$. Let $i^{(b)}$ be $i_{q-1}\dots i_{b+1}\bar{i}_b i_{b-1}\dots i_0$, where $\bar{i}_b$ is the complement of i_b and $0 \leq b < q$. A processor indexed i is directly connected to a processor indexed $i^{(b)}$, $0 \leq b < q$. Obviously, each processor has q neighbours in a hypercube of p processors.

In this paper we consider the hypercube model that consists of $N = n^3$ processors where $N = 2^q$. Conceptually, these processors may be regarded as arranged in an $n \times n \times n$ array pattern. Assume that the processors are indexed in row-major mode, that is, the processor positioned at (i, j, k), denoted by $PE_{i,j,k}$, has an index $in^2 + jn + k$. Thus, if $r_{3q-1}\dots r_{2q}r_{2q-1}\dots r_q r_{q-1}\dots r_0$ is the binary representation of $PE_{i,j,k}$, then $i = r_{3q-1}r_{3q-2}\dots r_{2q}$, $j = r_{2q-1}r_{2q-2}\dots r_q$, and $k = r_{q-1}r_{q-2}\dots r_0$. We further impose on this model the same restrictions that were imposed on the 2-D mesh array. That is, each processor has only a constant number of registers, each of which can store a word of $c \log n$ bits. The arithmetic operations and Boolean operations in all processors are synchronized.

3. The Algorithms on the 2-D Mesh Array

3.1 Finding the Most Vital Edge w.r.t. the MST

3.1.1 An Algorithm on the PRAM

In the following we reproduce the NC algorithm for the most vital edge problem w.r.t. MSTs on the PRAM [5].

Without loss of generality, assume that the weight on each edge is distinct. Let T be the MST of $G(V, E, w)$ and $E(T)$ be the edge set of T. We define the *replacement edge* $r(e)$ of an edge $e \in E(T)$ as follows. Deleting an edge $e = (u, v)$ from T divides T into two subtrees, T_1 and T_2. Let W be the vertex set of T_1 and $V - W$ be the vertex set of T_2. Let $Q = (W \times (V - W)) \cap E - \{e\}$ and e' be a non-tree edge such that $w(e') = \min\{w(e'') \mid e'' \in Q\}$. Define $r(e) = e'$ and $r(e)$ as unique.

Lin *et al.* [7] observed that the most vital edge must be in T. Iwano *et al.* [2] further showed that an edge $e^* \in E(T)$ is the most vital edge if and only if it satisfies the following equation:

$$w(e^*) = \max\{w(r(e)) - w(e) \mid e \in E(T)\} \qquad (1)$$

Obviously, e^* can easily be obtained by computing $r(e)$ for all $e \in E(T)$ in parallel. Thus, the NC algorithm depends on how to compute $r(e)$ for every $e \in E(T)$ efficiently. Iwano *et al.* have proven the following lemma [2].

Lemma 1. Let T be an MST of $G(V, W, w)$. Define an edge weight function $new()$ such that $new(e) = 0$ for $e \in E(T)$, and $new(e) = B - w(e)$ for $e \notin E(T)$ where $B = \max\{w(e) \mid e \in E\} + 1$. Let $MaxST(G)$ be a maximum spanning tree of G w.r.t. the weight function $new(e)$ and $E(MaxST(G))$ be the edge set of $MaxST(G)$. For an edge $e \in E(T)$, there exists a replacement edge $r(e)$ such that $r(e) \in E(MaxST(G))$, that is, $T - \{e\} \cup \{r(e)\}$ is a minimum spanning tree of $G - \{e\}$.

Lemma 1 suggests an algorithm for computing all $r(e)$ from the tree $MaxST(G)$. Let T_2 be the MST (MSF) of the graph $G(V, E - E(T), w)$ and $E(T_2)$ be the edge set of T_2. It is not difficult to show that $MaxST(G) = T_2$. We now are ready to compute $r(e)$ for each $e = (u, v) \in E(T)$. Assume that T is a rooted tree. For each vertex v, the preorder numbering $pre(v)$ and the number of descendants $nd(v)$ (including v itself) have been computed. Suppose that v is the parent of u in T; we have the following formula:

$$w(r(e)) = \min\{w(e'') \mid e''$$

$$= (u', v') \in E(T_2),\ pre(u) \leq pre(u') < pre(u) + nd(u),$$

$$\text{and either } pre(v') < pre(u) \text{ or } pre(v') \geq pre(u) + nd(u)\} \qquad (2)$$

By (1) and (2), we can easily find e^*, the most vital edge of G w.r.t. the MST T.

3.1.2 An Implementation on the 2-D Mesh Array

In the following we show how to realize the above approach on the 2-D mesh array. We list a few operations or subroutines on this model that will be used repeatedly later.

1. Horizontal rotation [16]. Every processor creates a "duplicate" copy of its contents, and then all the duplicate data items move like a row of soldiers: every row moves left, its left-most elements "bouncing back" on its left-most processor $PE_{i,0}$. The data that bounced in $PE_{i,0}$ move right until they bounce again, this time on the right-most processor $PE_{i,n-1}$; then they move left again until they go back to their initial position. It is clear that the data initially in every $PE_{i,j}$ visit all the other processors of row i. This operation takes $O(n)$ time.

2. Vertical rotation [16]. This is analogous to the previous data movement, but the data move vertically. This operation obviously takes $O(n)$ time, and the data initially in every processor $PE_{i,j}$ visit all the other processors in column j.

Using horizontal rotation and vertical rotation, Atallah *et al.* [16] have shown that all pairs, shortest paths and the transitive closure of $G(V, E, w)$ can be computed in $O(n)$ time on this model.

Let A and A^* be the adjacency matrix and the transitive closure matrix of G, respectively. Assume that entry $a_{i,j}$ of A is stored in $PE_{i,j}$ and entry $a^*_{i,j}$ of A^* is also stored in $PE_{i,j}$ for all i, j, $0 \leq i, j \leq n-1$. Then:

Lemma 2 [16]. Let A be the adjacency matrix of $G(V, E, w)$; the transitive closure matrix A^* of G can be computed in time $O(n)$ on a 2-D mesh array.

Similarly, all pairs' shortest paths of G also can be computed in $O(n)$ time on this model.

3. Random access write [17]. In this formulation, an index (i', j') is contained in $PE_{i,j}$. The data from a specified register of $PE_{i,j}$ are to be transmitted to $PE_{i',j'}$, $0 \leq i, i', j, j' \leq n-1$. It is assumed that no two processors are sending their data to the same destination, and if the destination address is ∞, then the data from $PE_{i,j}$ are not transmitted to any other processor. The details of how this data movement is implemented in $O(n)$ time can be found in [17].

4. Horizontal broadcasting. Assume that in each row i, an element a_i is stored in register X of some processor. The objective is to broadcast a_i to register X of all other processors in row i for all i, $0 \leq i \leq n-1$. Obviously this can be done in $O(n)$ time.

5. Vertical broadcasting. This is similar to horizontal broadcasting. The data initially in a processor $PE_{i_0,j}$ are sent to all the other processors of column j in $O(n)$ time.

6. Horizontal prefix. In this formulation, processor $PE_{i,j}$ contains an element $d_{i,j}$ that is stored in register D. Thus, the elements in row i form a sequence $d_{i,0}, d_{i,1}, \ldots, d_{i,n-1}$. The problem is to compute all prefixes of the sequence, that is, processor $PE_{i,j}$ will contain the result $\sum_{k=0}^{j} d_{i,k}$ in register $S[i, j]$ for all i, $0 \leq i \leq n-1$. We proceed as follows. First, all elements in column 0 set a tag "1" and $S[i, j] := d_{i,j}$ initially. Then, do $n-1$ rounds of the following routine. In this routine, each processor sends the content of S and the tag to its right neighbour provided that the right neighbour exists. If the tag of the sender is "1", then it sets its tag to "0" after finishing sending. When a processor receives a message with tag "1", it sets $S[i, j] := S[i, j] + s$, and sets its tag as "1", where s is the value just received. Obviously, the operations can be done in $O(n)$ time.

7. Vertical Prefix. This is similar to *horizontal prefix*, and so is omitted.

Having the subroutines above, we have the following lemmas.

Lemma 3. Suppose $a_0, a_1, \ldots, a_{n-1}$ are n numbers stored in n different processors in the 2-D mesh array, and are to be routed to n distinct destinations. Routing these elements to their destinations needs $O(n)$ time.

Proof. First, if a processor $PE_{i',j'}$ contains a_i, it forms a routing record $(PE_{i',j'}, PE_{des(a_i)}, a_i)$; otherwise, it forms a routing record $(PE_{i,j}, \infty, 0)$, where $PE_{des(a_i)}$ is the destination of a_i. Then, route these records to their destinations by random access write. Routing takes $O(n)$ time. The lemma follows. QED

Lemma 4. Let $a_0, a_1, \ldots, a_{n-1}$ be stored in some row (column) i on the 2-D mesh array, where a_j is stored in $PE_{i,j}$. Broadcasting a_i to the ith row and to the ith column needs $O(n)$ time for all i, j, $0 \leq i, j \leq n-1$.

Proof. First, each processor $PE_{i,j}$ forms a routing record $(PE_{i,j}, PE_{i,i}, a_i)$ if a_i is stored in it, and forms a routing record $(PE_{i,j}, \infty, 0)$ otherwise. Then, route these records to their destinations by random access write. As a result, a_i is in $PE_{i,i}$. Finally, the processors in each row (column) duplicate a_i to all processors by horizontal broadcasting (vertical broadcasting). Routing and broadcasting take $O(n)$ time respectively. The lemma follows. QED

We now consider the vital edge problem w.r.t MST. Assume that the adjacency matrix A of $G(V, E, w)$ is already stored in the 2-D mesh array, that is, entry $a_{i,j}$ of A is stored in register $A[i, j]$ of $PE_{i,j}$ for all i, j, $0 \leq i, j \leq n-1$.

Step 1. Compute the MST of $G(V, E, w)$. Let T be the MST of G. T is stored in an adjacency matrix T where entry $T[i, j] = T[j, i] = 1$ if (i, j) is an edge in T, and $T[i, j] = T[j, i] = 0$ otherwise. This step can be implemented in $O(n)$ time by Maggs *et al.*'s algorithm [18].

Step 2. Make T a rooted directed tree T_{in} in which the directions of all edges are pointed to the root. We select a vertex 0 to be the root of T_{in}. We also construct another directed tree T_{out} rooted at vertex 0 from T in which the directions of all edges leave from the root. This step is implemented by the following substeps.

Step 2.1. Compute all pairs shortest paths for T. Let T^* be the adjacency matrix of T where entry $T^*[i, j]$ stores the distance between i and j in T. By Lemma 2, this step can be done in $O(n)$ time.

Step 2.2. Broadcast the value of $T^*[0, i]$ to register $D_1[i, j]$ of $PE_{i,j}$ in the ith row and to register $D_2[j, i]$ of $PE_{j,i}$ in the ith column for all i, j, $0 \leq i, j \leq n-1$. This takes $O(n)$ time, by Lemma 3. Note that $T^*[0, i]$ is the level number of vertex i in T rooted at vertex 0.

Step 2.3. Construct the tree T_{in}. Every $PE_{i,j}$ runs the following instruction: if $T[i,j] = 1$ and $D_1[i,j] = D_2[i,j]-1$, then i is the parent of j in T_{in}, set $T_{in}[i,j] := 1$ and $F(j) := i$. Otherwise, $T_{in}[i,j] := 0$. The directed tree T_{out} can be constructed similarly. This step takes $O(1)$ time.

Step 3. Test whether there exist some edges in T_{in} that are the bridges of G using the algorithm in [16]. If yes, then any one of these edges is the most vital edge of G; stop. Otherwise, we continue the following steps. This step takes $O(n)$ time.

Step 4. Given T_{in}, compute the pre-order numbering $pre(v)$, the number of ancestors $na(v)$, and the number of descendants $nd(v)$ for every vertex $v \in V$, as follows.

Step 4.1. Compute the number of descendants $nd(i)$ for each $i \in V$. First, compute the transitive closure T^*_{out} of T_{out} using its adjacency matrix. Then for each row i, $0 \le i \le n-1$, compute $\sum_{j=0}^{n-1} T^*_{out}[i,j]$ and store this value to register $ND[i,0]$ of $PE_{i,0}$. It is clear that $nd(i) = \sum_{j=0}^{n-1} T^*_{out}[i,j] + 1$ because i is a descendant of itself by our definition. This step takes $O(n)$ time.

Step 4.2. Broadcast $ND[i,0]$ to all $PE_{i,j}$ in the ith row and store it in $ND_1[i,j]$; also broadcast it to all $PE_{j,i}$ in the ith column and store it in $ND_2[i,j]$, for all i,j, $0 \le i,j \le n-1$. This step takes time $O(n)$, by Lemma 4.

Before we continue, define an ordered tree T as a rooted tree in which all children $v_1, v_2, \ldots, v_l$ of an internal vertex u are ordered, that is, v_i is the ith child of u, $F(v_i) = u$, $1 \le i \le l$. For an ordered tree, Tsin and Chin [19] gave the following lemma.

Lemma 5. Let $T(V,E')$ be an ordered tree. For each $i \in V$:

$$pre(i) = \sum_{j \in ANC(i)} \sum_{w \in EBRO(j)} nd(w) + na(i) \qquad (3)$$

where $ANC(i)$ is the set of ancestors of i, $EBRO(j)$ is the set of elder brothers of j, and $na(i)$ is the number of ancestors of i.

Step 4.3. In order to finish step 4, T_{in} needs to be an ordered tree. To do so, we notice that in the ith row, if $T_{in}[i,j] = 1$, then j is one of children of i. The ranks (orders) of the children of i can be computed using the prefix computation. The rank of j is stored in register $Rank[i,j]$ of $PE_{i,j}$ if j is a child of i. Otherwise $Rank[i,j] := 0$.

Step 4.4. Now, T_{in} is an ordered tree. By (3) we can compute
$\sum_{w \in EBRO(j)} nd(w)$ for every j. If $T_{in}[i,j] = 1$ (i is the parent of j), then the number of the elder brothers of j in T_{in} $Ebro[i,j] := \sum_{k=0,\ T_{in}[i,k]=1}^{j-1} ND_2[i,k]$; otherwise $Ebro[i,j] := 0$. Obviously, this can be done by a prefix computation. Note that only one value of $Ebro[i,j]$ in column j is nonzero because j has a unique parent. We can route this value to $PE_{0,j}$, that is, $Ebro[0,j] := Ebro[i,j]$ if $T_{in}[i,j] = 1$.

Step 4.5. We now compute $pre(i)$ for each $i \in V$ in T_{in}, by the formula $pre(i) = \sum_{j \in ANC(i)} \sum_{w \in EBRO(j)} nd(w) + na(i) = \sum_{j \in ANC(i)} (\sum_{w \in EBRO(j)} nd(w) + 1)$ for every vertex i. First, compute the transitive closure T^*_{in} of T_{in}. Then, consider the ith column: if $T^*_{in}[j,i] = 1$ (j is an ancestor of i), processor $PE_{j,i}$ sets $Temp[j,i] := Ebro[j,i]+1$ where the additional term "1" indicates that j is an ancestor of i; otherwise set $Temp[j,i] := 0$. Compute all prefixes of the sequence $Temp$ for each column i. Let $Fix(j,i) = \sum_{k=0}^{j} Temp[k,i]$. Obviously, $pre(i) = Fix[n-1,i] = \sum_{j \in ANC(i)} \sum_{w \in EBRO(j)} nd(w) + na(i)$. Finally, send $pre(i)$ to register $PRE[i,0]$ of $PE_{i,0}$ by random access write.

Step 5. Broadcast $pre(i)$ to register $PRE_1[i,j]$ of $PE_{i,j}$ in the ith row, and register $PRE_2[j,i]$ of $PE_{j,i}$ in the ith column. As results, $PE_{i,j}$ has $pre(i)$ and $pre(j)$ for all i,j, $0 \le i,j \le n-1$.

Step 6. Construct the MST T_2 of graph $G(V, E - E(T), w)$, using Maggs *et al.*'s algorithm. The adjacency matrix A' of this graph can be obtained in $O(1)$ time by testing whether an edge (i,j) belongs to T. If it does, set $A'[i,j] := 0$. Otherwise, $A'[i,j] := A[i,j]$, where A is the weighted adjacency matrix of $G(V,E,w)$.This step can be implemented in time $O(n)$ on this model. After this, all the edges in T_2 are directed to the root.

Step 7. Construct a routing record in every processor $PE_{i,j}$. If (i,j) is an edge of T_2 and i is the parent of j, then form a record $(i,j,pre(i),pre(j),w(i,j),PE_{0,j})$ where $PE_{0,j}$ is the destination address of the record; otherwise, form a record $(i,j,pre(i),pre(j),\ w(i,j),\infty)$. Note that $pre(i) = PRE_1[i,j]$ and $pre(j) = PRE_2[i,j]$ are available in $PE_{i,j}$. Routing the records to their destinations takes $O(n)$ time [17]. After the routing, the record $(i,j,pre(i),pre(j),\ w(i,j),PE_{0,j})$ is stored in $PE_{0,j}$ as follows. The edge (i,j) is stored in register $T_2[0,j] := (i,j)$, $pre(i)$ and $pre(j)$ are stored in registers $PT_1[0,j]$ and $PT_2[0,j]$, and the weight $w(i,j)$ of (i,j) is stored in register $W_2[0,j]$. As a result, the edges in T_2 along with the pre-order numberings of their endpoints in T_{in} are stored in the first row of the mesh array.

Step 8. Broadcast each record in the first row to the column in which that record is located, and store the corresponding components of the record to the registers of the processors in that column. This can be done in $O(n)$ time. Now T_2 is stored in each row.

Step 9. Find the most vital edge using (1). The details are as follows.

Step 9.1. If $T_{in}[i,j] = 1$ (i is the parent of j), $PE_{i,j}$ forms a routing record $(i,j,pre(i),pre(j),w(i,j),PE_{j,0})$ where $PE_{j,0}$ is the destination address of the record; otherwise, it forms a routing record $(i,j,pre(i),pre(j),\ w(i,j),\infty)$. Route all records to their destinations. After the routing, the record $(i,j,pre(i),pre(j),\ w(i,j),PE_{j,0})$ is stored in $PE_{j,0}$ as follows: the edge (i,j) is stored in register $T_1[j,0] := (i,j)$; $pre(i)$ and $pre(j)$ are stored in registers $PR_1[j,0]$ and $PR_2[j,0]$; and the weight $w(i,j)$ of (i,j) is stored in register $W_1[i,0]$. The routing costs $O(n)$ time. As a result, all the edges in T_{in} are stored in the first column. Broadcast $PR_1[i,0]$ and $PR_2[i,0]$ to the ith row, for all i, $0 \le i \le n-1$.

Step 9.2. Each $PE_{i,j}$ checks (2). If "yes", the edge (x,j) in T_2 (x is the parent of j in T_2) is a candidate for replacing the edge (y,i) in T where y is the parent of i in T. This step takes $O(1)$ time.

Step 9.3. Each $PE_{i,j}$ who checked "yes" in Step 9.2 reports the weight of edge (x, j) to $PE_{i,0}$. So, $PE_{i,0}$ can determine the minimum weighted candidate, which is the replacement for (y, i) of T. Then, compute the difference $w(x, j) - w(y, i)$ and store it in $W[i, 0]$ of $PE_{i,0}$. This step takes $O(n)$ time.

Step 9.4. For each $PE_{i,0}$, it first sets $W[i, 0] := W[i, 0] - W_1[i, 0]$. Then, by (1), find the maximum value which is not ∞ from all $W[i, 0]$ in column 0, $0 \leq i \leq n-1$. The corresponding edge in T_{in} is the most vital edge.

From the above algorithm, we have the following theorem.

Theorem 1. Let $G(V, E, w)$ be an undirected, weighted graph, finding the most vital edge w.r.t. MSTs needs $O(n)$ time on an $n \times n$ mesh array.

3.2 Finding the Most Vital Edge w.r.t. the SSP Problem

In this subsection we deal only with an undirected, weighted graph $G(V, E, w)$. Consider the most vital edge problem w.r.t. a single-source shortest path between vertices s and t.

3.2.1 A Sequential Algorithm

Before we proceed, let us recall the algorithm given by Malik *et al.* [11] for the problem. Let T_s be a shortest path tree starting at s and $\pi_{s,t}$ be the shortest path from s to t in T_s. The tree T_t can be defined similarly. Let $d_s(i)$ and $d_t(i)$ be the lengths of the shortest paths from s to i in T_s and from t to i in T_t, respectively, where $i \in V$. If an edge (i, j) in $\pi_{s,t}$ is removed, the vertex set V is divided into two-disjoint subsets W and $V - W$ where W contains all vertices reachable from s in T_s and $V - W$ contains all vertices reachable from j including t. Define:

$$Q(i, j) = \{(x, y) \mid (x, y) \in E,\ x \in W,\ y \in V - W\} \quad (4)$$

The correctness of their algorithm is based on the following two important observations.

Observation 1 [11]. An edge (i, j) is in $\pi_{s,t}$ if and only if $d_s(t) = d_t(s) = d_s(i) + w(i, j) + d_t(j) = \min_{(x,y)\in E}\{d_s(x) + w(x, y) + d_t(y)\}$ where i is the immediate predecessor of j in $\pi_{s,t}$.

Observation 2 [12]. If some edge $(i, j) \in \pi_{st}$ is removed from T_s, dividing the vertex set V into V_s and $V - V_s$ such that $s \in V_s$ and $t \in V - V_s$ where V_s contains all vertices reachable from s, then there exists shortest paths from all other vertices in $V - V_s$ to t that do not use the edge (i, j).

Note that the original Observation 2 in [11] is incorrect, as was shown by a counterexample by Bar-Noy *et al.* [12]. Also, note that if the most vital edge is unique, then it must be in $\pi_{s,t}$; otherwise, there is at least one in $\pi_{s,t}$ [14]. In our algorithm we are interested in finding the one in $\pi_{s,t}$. We only consider the effect of removal of each edge in $\pi_{s,t}$ on the length of the shortest path from s to t. As $Q(i, j)$ is a cut between s and t, any shortest path from s to t in graph $G' = (V, E - \{(i, j)\}, w)$ must have an edge in $Q(i, j)$. By Observations 1 and 2, the length of the shortest path from s to t in G' via such an edge can be computed by the following formula:

$$\min_{(x,y)\in Q(i,j)}\{d_s(x) + w((x, y)) + d_t(y)\} \quad (5)$$

Therefore, if both T_s and T_t are given, the most vital edge can be found in time $O(m|\pi_{s,t}|)$ by checking every edge in $\pi_{s,t}$ using formula (5). By exploring the special property of undirected graphs, Malik *et al.* [11] suggested an $O(m + n \log n)$ time algorithm for this problem using Fibonacci heap technique due to Fredman *et al.* [20]. However, their algorithm is highly sequential; a direct implementation of their algorithm on an $n \times n$ mesh needs $O(n^2)$ time.

In the following we first propose an inefficient sequential algorithm for this problem. We then show the proposed algorithm can be easily implemented in $O(n)$ time on the 2-D mesh array, which is time optimal on this model. The correctness of the proposed algorithm is also based on the two observations above.

We assign the edges in G with a new weight function w_1 that is defined as follows. If an edge $e = (u, v) \in E(T_s)$, set $w_1(e) := +\infty$. If $e \notin E(T_s)$, we classify it into three cases: (1) If t is a descendant of v, or v is a descendant of t in T_s but u is not, then $w_1(e) := d_t(v) + d_s(u) + w(u, v)$. (2) If t is a descendant of u, or u is a descendant of t in T_s but v is not, then $w_1(e) := d_t(u) + d_s(v) + w(u, v)$. (3) Otherwise $w_1(e) := +\infty$. $w_1(e)$ is actually the shortest length from s to t via e.

Given two vertices u and v, testing whether they have an ancestor-descendant relationship in T_s can be done in constant time as follows. We first traverse T_s by assigning a pre-order numbering $pre(v)$ and computing the number of descendants $nd(v)$ for each vertex v in T_s. We then test whether $pre(u) \leq pre(v) < pre(u) + nd(u)$ holds. If yes, v is a descendant of u and u is an ancestor of u in T_s; otherwise, they do not have any ancestor–descendant relationship.

Having done the above, we now compute the MST of graph $G(V, E, w_1)$ with the new weight function w_1. Let T_2 be the MST of this graph. Consider the tree T_s, and define the replacement edge $r(e)$ for each edge $e \in T_s$ as follows. The vertex set V of T_s is divided into two subsets V_s and V_t that contain s and t respectively, after the removal of e from T_s. Let $e' \in (V_s \times V_t) \cap E$ be an edge such that $w_1(e') = \min\{w_1(e'') \mid e'' \in (V_s \times V_t) \cap E - \{e\}\}$. Define $r(e) = e'$. If there exists no edge between V_s and V_t, then e is a bridge of G. Therefore, it is the most vital edge of G. For convenience, we set $r(e) = +\infty$ for every edge $e \in \pi_{s,t}$ initially.

Lemma 6. Let $e \in \pi_{s,t}$ and $r(e)$ be defined as above. Then $\{r(e) \mid e \in \pi_{s,t}\} \subseteq E(T_2)$.

Proof. Trivial. QED

Lemma 7. An edge $e^* \in \pi_{s,t}$ is the most vital edge of $G(V, E, w)$ w.r.t SSP if $w_1(r(e^*)) = \max\{w_1(r(e)) \mid e \in \pi_{s,t}\}$.

Proof. Let $e^* = (u, v)$ where u is the immediate predecessor of v in $\pi_{s,t}$. Let $r(e^*) = (u', v')$. By the definition, $w_1(r(e^*)) = d_s(u') + d_t(v') + w(u', v')$ and $w_1(r(e^*)) =$

$\max\{w_1(r(e)) \mid e \in \pi_{s,t}\}$. By Observations 1 and 2, the lemma follows. QED

3.2.2 An Implementation on the 2-D Mesh Array

From the discussion above, we see that all the other operations are similar to those used for the most vital edge problem w.r.t. MSTs in Section 3.1. We omit them here. However, the construction of the tree T_s is different from that for the MST T of $G(V, E, w)$ in Section 3.1. In the following we show how to construct T_s. The construction of T_t is similar, and so is omitted. We proceed as follows.

Step 1. Compute all pairs' shortest paths in G. Denote by S the resulting distance matrix. Then $d_s(i) = S[s, i]$ and $d_t(i) = S[t, i]$, obviously.

Step 2. Broadcast $d_s(i)$ to the ith row and the ith column, that is, $D_s^{(1)}[i, j] := d_s(i)$ and $D_s^{(2)}[j, i] := d_s(j)$ for all j, $0 \leq j \leq n-1$.

Step 3. Construct a directed version of T_s, a directed tree T_s^{in} rooted at s in which the direction of all edges are pointed to the root, which is described as follows. Each $PE_{i,j}$ runs the following instruction: if $d_s(i) + w(i, j) = d_s(j)$, set $PE_{i,j}$'s mask "1"; otherwise set its mask "0". In the ith column, select a j_0 such that $j_0 = \min\{j \mid PE_{j,i}$ *is masked by* "1"$\}$; set $F(i) := j_0$ and $T_s^{in}[j_0, i] := 1$. Meanwhile, a routing record in the form as (*the address of the source, the destination address, label*) is constructed for later use, where if *label* $= 1$, the edge (j_0, i) is in T_s, and otherwise (j_0, i) is not in T_s. Here the record is $(PE_{j_0,i}, PE_{i,j_0}, 1)$.

After Step 3, T_s^{in} has been established. Now consider constructing the directed tree T_s^{out} in which the direction of all edges leave from the root. It can be easily implemented by routing the records formed in Step 3: if $PE_{i,j}$ gets a message from $PE_{j,i}$ with *label* $= 1$, set $T_s^{out}[j, i] := 1$; otherwise set $T_s^{out}[j, i] := 0$. Note that T_s^{in} and T_s^{out} correspond to the trees T_{in} and T_{out} in Section 3.1.2, respectively.

Compute $pre(v)$ and $nd(v)$ for each vertex $v \in V$ by pre-order transversal on T_s^{in}, broadcast $pre(v)$ and $nd(v)$ along the vth row and the vth column, and broadcast $pre(t)$ and $nd(t)$ to all processors on the 2-D mesh array. All these operations can be done in $O(n)$ time, by the discussion in Section 3.1.

Next, consider assigning the new weight function w_1 to all the edges in $G(V, E)$. Having the trees T_s^{in} and T_t^{in}, broadcast $d_s(v)$ and $d_t(v)$ along the vth row and the vth column, respectively. Now for an edge $e = (i, j)$, if it is an edge of T_s^{in}, which can be easily checked, then assign $w_1(i, j) := w_1(j, i) := +\infty$. Otherwise, first check whether t is a descendant of i in T_s^{in}; this can be done by checking whether $pre(i) \leq pre(t) < pre(i) + nd(i)$ holds. If it does, then t is a descendant of i. Otherwise i is not a descendant of t. To check whether j is a descendant of t can be done similarly. The testing can be finished in $O(1)$ time because all data needed have already been stored in processor $PE_{i,j}$.

By the result in Section 3.2.1, we know that the remaining part is to find the replacement edges in graph $G(V, E, w_1)$. The operations are similar to those used in Section 3.1, and are omitted. Therefore, we have:

Theorem 2. Given an undirected simple graph $G(V, E, w)$ with n vertices, and two specified vertices s and t, the most vital edge w.r.t. a shortest path between s and t can be found in $O(n)$ time on an $n \times n$ mesh array.

4. The Algorithms on the Hypercube Array

Let us recall the hypercube array defined in Section 2. Each processor in this array is indexed in the form of (i, j, k), where $0 \leq i, j, k < n$. Now we fix one parameter of the three parameters i, j, and k, for example, $i = 0$. Then, all processors $PE_{0,j,k}$ and their interconnection structure, $0 \leq j, k < n$, form a sub-hypercube $\mathcal{Q}$, and $\mathcal{Q}$ may be regarded as a 2-D array in which the indexing scheme is the same as that for the 2-D mesh array but the connection structure among processors is different.

From the algorithms in Section 3, we know that these algorithms involve the transitive closure computation which serves as a key subroutine. Dekel *et al.* [21] have shown how to implement this computation on the hypercube array in $O(\log^2 n)$ time by showing that a matrix multiplication $C = A \times B$ can be done in $O(\log n)$ time on this model. Assume that the data distribution of A, B, and C in the hypercube array is as follows:

$$A[0, j, k] := a_{j,k}$$

$$B[0, j, k] := b_{j,k}$$

$a_{j,k}$ and $b_{j,k}$ are the elements of the two matrices to be multiplied, $0 \leq j, k < n$. The desired final result C is $C[0, j, k] := c_{j,k} = \sum_{l=0}^{n-1} a_{j,l} b_{l,k}$, $0 \leq j, k < n$.

Let A be the adjacency matrix of G with n vertices and A^* be the transitive closure of G. It is well known that $A^* = A + A^2 + \ldots + A^n$. Suppose that the matrices A and A^* are stored in $\mathcal{Q}$, that is, entry $a_{i,j}$ of A and entry $a^*_{i,j}$ of A^* are stored in registers $A[0, i, j]$ and $A^*[0, i, j]$ of $PE_{0,i,j}$, respectively, $0 \leq i, j < n$. It is easy to derive the following lemma.

Lemma 8 [21]. Let the adjacency matrix A of G be stored initially on $\mathcal{Q}$ — a sub-hypercube in the hypercube array. The transitive closure A^* of G can be computed, and finally stored in $\mathcal{Q}$ in $O(\log^2 n)$ time by the hypercube array.

Dekel *et al.* also pointed out that all pairs' shortest paths of G can be computed and finally stored in $\mathcal{Q}$ within $O(\log^2 n)$ time.

Note that only the transitive closure computation and all the pairs' shortest path computations involve all processors in the hypercube array, but all the other steps of the algorithms in Section 3 can be simulated on $\mathcal{Q}$ in $O(\log^2 n)$ time.

The data movement on this model can be dealt with similarly. As mentioned before, we may think of $\mathcal{Q}$ as a 2-D array in which the processors are hypercube-connected, the processors $PE_{0,i,j}$ with $0 \leq j < n$ form row i, and the processors $PE_{0,k,j}$ form column j, $0 \leq k < n$. The data movements on $\mathcal{Q}$ are described as follows.

As both horizontal rotation and vertical rotation operations are used for computing the transitive closure and all pairs' shortest paths, it is well known that they take $O(\log^2 n)$ time using n^3 hypercube-connected processors [21]. Now we discuss the other data movements on this model.

1. Random access write [17]. In this formulation, an index $(0, i', j')$ is contained in $PE_{0,i,j}$. The data from a specified register of $PE_{0,i,j}$ are to be transmitted to $PE_{0,i',j'}$, $0 \le i, i', j, j' \le n-1$. It is assumed that no two processors are sending their data to the same destination, and if the destination address is ∞, then the data from $PE_{0,i,j}$ are not transmitted to any processor. The details of how this data movement is implemented in $O(\log^2 n)$ time on $\mathcal{Q}$ can be found in [17].

2. Horizontal broadcasting. Assume that in each row i, an element a_i is stored in register X of some processor, the objective is to broadcast a_i to register X of the other processors in row i for all i, $0 \le i \le n-1$. As each row itself is a q-dimensional hypercube, it is well known that broadcasting an element to all processors of a q-dimensional hypercube takes $q = \log n$ time. Therefore, broadcasting an element in some processor in row i to all the other processors at that row takes $O(\log n)$ time.

3. Vertical broadcasting. This is the vertical analog of horizontal broadcasting.

4. Horizontal prefix. In this formulation, each processor $PE_{0,i,j}$ contains an element $d_{0,i,j}$ stored in register D locally. Thus, the elements in row i form a sequence $d_{0,i,0}, d_{0,i,1}, \ldots, d_{0,i,n-1}$. The problem is to compute all prefixes of the sequence, that is, processor $PE_{0,i,j}$ will contain the result $\sum_{k=0}^{j} d_{0,i,k}$, which is stored in register $S[0,i,j]$ for all i, $0 \le i \le n-1$. Because the processors in row i and their interconnection structure form a sub-hypercube, the problem now becomes one of computing all prefixes in a hypercube. In the following we present an algorithm for this purpose. Our algorithm is directly derived from the algorithm for the polynomial evaluation problem on a prefect shuffle network of n processors by Stone [22]. Let $j_{q-1}j_{q-2}\ldots j_0$ be the binary representation of j. The prefix computation in a q-dimensional hypercube can be described as a recursive process. A q-dimensional hypercube consists of two $(q-1)$-dimensional hypercubes, C_0 and C_1, where C_0 contains all processors with the highest bit "0" in their addresses, and C_1 contains all processors with the highest bit "1" in their addresses. If we proceed with the prefix computation in each hypercube independently and simultaneously, then the results in C_0 are the correct results. For C_1, the results are not correct; we need to add the largest prefix, which is in the processor indexed by $0\underbrace{111\ldots111}_{q-1}$. Thus, the algorithm is as follows:

`Algorithm Prefix`(q);
/* q is the number of dimensions of the hypercube */
if if $q = 0$ then
every PE_j with $j_0 = 0$ sends the number to its neighbour $PE_{j'}$ with $j'_0 = 1$, and every $PE_{j'}$ with $j'_0 = 1$ receives a number from its neighbour PE_j with $j_0 = 0$ and adds this number to its own number.
endif;
if $q > 0$ then
(1) Call `Prefix`$(q-1)$;
(2) Each PE_j with $j = ****0\underbrace{1111\ldots111}_{q-1}$ sends the result to its neighbour $PE_{j'}$ with $j'_{q-1} = 1$;
(3) The q-dimensional hypercube is formed by the right-most q bits $****j_{q-1}j_{q-2}\ldots j_0$.
Broadcast the value received from PE_{j_0} with $j_0 = ****0\underbrace{111\ldots111}_{q-1}$ to all the processors $PE_{j'}$ with $j'_{q-1} = 1$ in this q-dimensional hypercube.
Each processor PE_j in this q-dimensional hypercube with $j_{q-1} = 1$ then adds the received value to its own value obtained in Step (1).
endif.

where "*" stands for either "1" or "0". By induction on q, it is easy to show that `Prefix` will compute all prefixes correctly. Therefore, the time used for computing all prefixes $T(q)$ is $T(q) = T(q-1) + O(q) = O(q^2) = O(\log^2 n)$.

5. Vertical prefix. This operation is similar to the horizontal prefix.

Having the routing subroutines above, we have the following lemmas.

Lemma 9. Let $a_0, a_1, \ldots, a_{n-1}$ be stored in n different processors on $\mathcal{Q}$— a sub-hypercube of the hypercube array. Along with a_i, $0 \le i \le n-1$, the destination processor address (also in $\mathcal{Q}$) is also stored in the same processor. Assume that all destinations are distinct. Then routing these elements to their destinations takes $O(\log^2 n)$ time.

Proof. First, if a processor $PE_{0,i',j'}$ contains a_i, it forms a routing record $(PE_{0,i',j'}, PE_{des(a_i)}, a_i)$; otherwise, it forms a routing record $(PE_{0,i,j}, \infty, 0)$, where $PE_{des(a_i)}$ is the destination of a_i. Then, route the records to their destinations by random access write. As results, a_i is sent to the $PE_{des(a_i)}$ for all i, $0 \le i \le n-1$. Routing takes $O(\log^2 n)$ time. The lemma follows. QED

Lemma 10. Let $a_0, a_1, \ldots, a_{n-1}$ be stored in some row (column) i on $\mathcal{Q}$ — a sub-hypercube of the hypercube array, where a_j is stored in $PE_{0,i,j}$. Broadcasting a_i to the ith row and to the ith column needs $O(\log^2 n)$ time for all i, j, $0 \le i, j \le n-1$.

Proof. Obvious and also omitted. QED

From the discussion above, we know that all steps in algorithms for the 2-D mesh array can be simulated on the hypercube array, using the data movement operations above, and any such simulation takes $O(\log^2 n)$ time at most. Therefore, we have:

Theorem 3. Given an undirected simple graph $G(V, E, w)$ with n vertices, the most vital edge w.r.t. either MST or SSP can be found in $O(\log^2 n)$ time using $n \times n \times n$ processors on a hypercube-connected processor array.

5. Conclusion

We have presented an $O(n)$ time algorithm for finding the most vital edge w.r.t. either MST or SSP on an $n \times n$ 2-D mesh array and an $O(\log^2 n)$ time algorithm on an $n \times n \times n$

hypercube array. The algorithm on the 2-D mesh array is time-optimal. The algorithm on the hypercube array is faster, than that a previously obtained [1]. Through the discussion, we know that the algorithm for the hypercube array is inefficient in terms of the amount of work. It raises an open problem, that is, whether there exists an algorithm for the most vital edge problem w.r.t. MST for a hypercube that requires $O(\log^2 n)$ time using only $n \times n$ rather than n^3 processors.

Acknowledgements

The research by Weifa Liang was partially supported by the Australian Research Council under a small grant schema (Grant No. F00025), and the research by Xiaojun Shen was partially supported by an NSF Grant No. C-CR-9810692.

References

[1] L. Hsu, R. Jan, Y. Lee, & C. Hung, Finding the most vital edge with respect to minimum spanning tree in a weighted graph, *Inform. Proc. Lett., 39*, 1991, 277–281.

[2] K. Iwano & N. Katoh, Efficient algorithms for finding the most vital edge of a minimum spanning tree, *Inform. Proc. Lett., 48*, 1993, 211–213.

[3] L. Hsu, P. Wang, & C. Wu, Parallel algorithms for finding the most vital edge with respect to minimum spanning tree, *Parallel Computing, 18*, 1992, 1143–155.

[4] H. Shen, *Improved parallel algorithms for the most vital edge of a graph with respect to minimum spanning tree*, Technical Report, Department of Computer Science, Abo Akademi University, Finland, 1995. URL: http://www.abo.fi/m̃ats/cs/publications/reports

[5] W. Liang & X. Shen, Finding k most vital edges in the minimum spanning tree problem, *Parallel Computing, 23*, 1997, 1889–1907.

[6] G.N. Frederickson & R. Solis-Oba, Increasing the weight of minimum spanning trees, *Proc. 7th Annual ACM-SIAM Symp. on Discrete Algorithms*, 1996, 1–8.

[7] K. Lin & M. Chern, The most vital edges in the minimum spanning tree problem, *Inform. Proc. Lett., 45*, 1993, 25–31.

[8] W. Liang & G. Havas, Finding the k most vital edges with respect to minimum spanning trees with $k = 2, 3$, *Proc. CATS'98, Computing Theory*, ed. X. Lin (Perth: Springer, 1998).

[9] H. Shen, *Finding the k most vital edges with respect to minimum spanning tree*, Technical Report, Department of Computer Science, Abo Akademi University, Finland, 1995. URL: http://www.abo.fi/m̃ats/cs/publications/reports

[10] H.W. Corley & D.Y. Sha, Most vital links and nodes in weighted networks, *Operations Research Letters, 1*, 1982, 157–160.

[11] K. Malik, A.K. Mittal, & S.K. Gupta, The k most vital arcs in the shortest path problem, *Operations Research Letters, 8*, 1989, 223–227.

[12] A. Bar-Noy, S. Khuller, & B. Schieber, *The complexity of finding most vital arcs and nodes*, CS-TR-3539, Department of Computer Science, University of Maryland, College Park, MD, U.S.A., 1995.

[13] M.O. Ball, B.L. Golden, & R.V. Vohra, Finding most vital arcs in a network, *Operations Research Letters, 8*, 1989, 73–76.

[14] W. Liang, X. Shen, & B.B. Zhou, Finding k most vital edges in the shortest path problem, unpublished ms.

[15] S. Venema, H. Shen, & F. Suraweera, NC algorithms for the single most vital edge problem with respect to the shortest path, *Inform. Proc. Lett., 60*, 1996, 243–248.

[16] M. Atallah & S.R. Kosaraju, Graph problems on a mesh-connected processor array, *J. ACM, 31*, 1984, 649–647.

[17] D. Nassimi & S. Sahni, Data broadcasting in SIMD computers, *IEEE Trans. Comput., C-30*, 1981, 101–106.

[18] B.M. Maggs & S.A. Plotkin, Minimum-cost spanning tree as a path-finding problem, *Inform. Proc. Lett., 26*, 1988, 291–293.

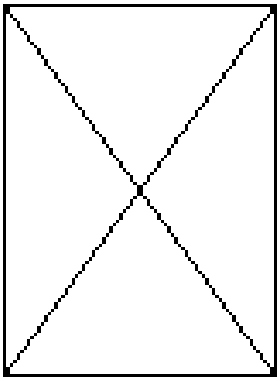
[19] Y.H. Tsin & F.Y. Chin, Efficient parallel algorithms for a class of graph theoretic problems, *SIAM J. Comput., 13*, 1984, 580–599.

[20] M.L. Fredman & R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM, 34*, 1987, 596–615.

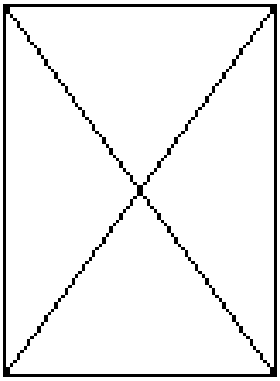
[21] E. Dekel, D. Nassimi, & S. Sahni, Parallel matrix and graph algorithms, *SIAM J. Comput., 10*, 1981, 547–675.

[22] H.S. Stone, Parallel processing with the perfect shuffle, *IEEE Trans. Comput., C-20*, 1971, 153–161.

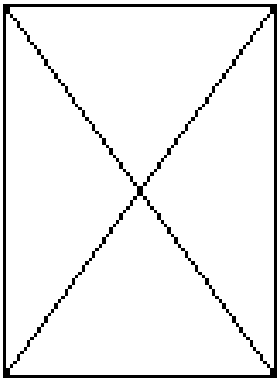
Biographies



Weifa Liang received his Ph.D. degree from the Australian National University in 1998, his M.E. from the University of Science and Technology of China in 1989, and his B.Sc. degree from Wuhan University, China, in 1984, all in computer science. He is currently a lecturer in the Department of Computer Science at the Australian National University. His major research interests include routing protocol design for high-speed networks, parallel processing, parallel and distributed algorithms, data warehousing and OLAP, query optimization, and graph theory.



Xiaojun Shen received his Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, U.S.A., in 1989. He received his M.Sc. degree in computer science from the Nanjing University of Science and Technology, China, in 1982, and his B.Sc. degree in numerical analysis from Qinghua University, Beijing, China, in 1968. He is currently an associate professor in the Computer Science Telecommunications Program at the University of Missouri—Kansas City, U.S.A. His current research interests include interconnection networks, algorithms, and computer networking.



Qing Hu received his B.Sc. and M.Sc. degrees in computer science from Nanjing University of Science and Technology (formerly East China Institute of Technology), China, in 1982 and 1989, respectively. He then was with the Computer Science Department of that university until 1991. He earned his Ph.D. degree in computer science from the University of Missouri–Kansas City in 1995. His current research interests include parallel processing, computer networking, image processing, and pattern recog-

nition. He is also working on distributed medical database systems.