On constructing multiple spanning trees in a hypercube

Feng-Hsu Wang and Ferng-Ching Lin

Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, ROC

Communicated by K. Ikeda
Received 3 February 1992
Revised 11 November 1992
Communicated by J. Håstad

Abstract


Simple formulas are derived to construct inorder spanning trees in a hypercube network. We identify a routing strategy to ensure the edge-disjointness of the routing paths in executing binary tree algorithms. Large trees can be built in a small hypercube by spreading the load congestion uniformly across the nodes of the hypercube. We also discuss the fault tolerance of the embedding method.

Keywords: Distributed data structure; communication tree; inorder spanning tree; edge congestion; node congestion; fault tolerance

1. Introduction

Tree-like data communication patterns are often used in multicomputers for designing efficient parallel algorithms, such as broadcast, prefix computation, etc. Due to the bountiful embedding capability of a hypercube, numerous communication trees embedded on a hypercube have been explored [1,5,6,10]. In this paper, we investigate how to construct multiple trees in a hypercube for the applications like segmented scan [2,7] and region growing [8]. In the segmented scan problem, the data set is divided into segments and a scan is performed on each segment of data. For the region growing problem, one essential operation is the merging of two consecutive regions. These applications are characterized by the data mapping scheme that preserves data adjacency for convenient neighborhood communication, and the concurrent tree operations of each data set. The tree operations we consider are special cases of a type of binary tree algorithms, called normal algorithms [4], in which only one level of nodes is active at a time.

We aim to propose a distributed data structure for the aforementioned problems. An inorder embedding method is especially useful for our purpose, which is derived from an earlier result on embedding a complete binary tree based on a grey code mapping scheme with unit expansion and a dilation of 2 [1,5]. We generalize the method to embed multiple binary trees, either complete or incomplete, on a hypercube. We first derive simple formulas with only a few addition and bit operations to build a complete inorder spanning tree. Then we present a procedure to build trees
Fig. 1. A binary 3-cube of arbitrary size. Each processor computes the formulas locally to set up the trees. A routing strategy is identified to make the data routing paths edge-disjoint for normal tree algorithms.

For a practical point of view, it is very important to address the issues of building large trees on a small hypercube and fault tolerance of the embedding method. Study of these issues on two tree embedding methods are presented in [14]. For our method, we show that the load congestion can be spread uniformly across the nodes of the hypercube. We also discuss the fault tolerance of the embedded trees.

2. Preliminaries

An n-cube is an n-dimensional hypercube network formed by \( N = 2^n \) processors. Each processor is represented by an n-bit number. Two processors are directly connected if and only if their binary representations differ in exactly one bit position. Figure 1 shows a 3-cube. Basic topological properties of the hypercube are discussed in [11]. In particular, we are interested in constructing a Hamiltonian path in the hypercube using a grey code indexing scheme. A grey code is a permutation of the ordered set \([0, 1, \ldots, N-1]\) such that the codes of two neighboring items have exactly one bit in difference. (The first and last items are considered neighbors.) There are many ways to generate grey code for different applications. A commonly used one is the Binary Reflected Grey Code (BRGC) [3]. Let \( G_n(x) \) denote the n-bit grey code of a number x. It is generated recursively starting with \( G_1 = [0, 1] \). From the grey code

\[
G_k = \left[ G_k(0), G_k(1), \ldots, G_k(2^k - 1) \right],
\]

the next grey code is given by

\[
G_{k+1} = \left[ 0G_k(0), 0G_k(1), \ldots, 0G_k(2^k - 1), 1G_k(2^k - 1), 1G_k(2^k - 2), \ldots, 1G_k(0) \right].
\]

In the following, unless stated otherwise, we assume the codes have n bits and write \( G_n \) as \( G \). To store an ordered set of \( N \) data items, one can map the \( i \)th data item to processor \( G(i) \) to preserve the data adjacency.

We will use the following notations to facilitate the presentation. Let \( a^{n-k}b^k \) denote the binary bit pattern formed by concatenating \( n-k \) a’s and \( k \) b’s, where \( a \) and \( b \) can be 0, 1, or * (don’t care). We use \( x_i \) to denote the \( i \)th bit of the binary number \( x \). We also use \( +, \cdot, \oplus, A \gg 1, A \ll 1 \) to represent the operations of bitwise OR, bitwise AND, bitwise exclusive-OR, shift one bit right and shift one bit left, respectively.

![Diagram](image-url)
Two useful lemmas about grey code are listed below.

**Lemma 1** [3]. \( G(x) = (x \oplus (x/2)) \) and \( x_i = G(x)_{n-1} \oplus G(x)_{n-2} \oplus \cdots \oplus G(x)_i \) for \( 0 \leq i \leq n - 1 \).

**Lemma 2** [3]. If \( i > 0 \) then \( G(x) \) and \( G(x + 2^i) \) differ in exactly two bits.

### 3. Embed inorder spanning trees

We first consider the problem of constructing a complete inorder spanning tree over the data set \([a_0, a_1, \ldots, a_{n-1}]\). Some assumptions are made here. Each data item is supposed to know its index within the ordered set, the processor address of the first data item, and the size of the set. Let \( s \) be the address of the starting processor, then the data item \( a_i \) is mapped to processor \( G(G^{-1}(s) + i) \). Figure 2 illustrates a complete inorder spanning tree to be constructed. In the tree, a node labelled \( i \) represents the processor to which the data item \( a_i \) is mapped. Depending on their functions, the nodes in the tree are categorized into three types. A *data node* is to store a data item. An *operation node* may perform computations or pass results. The operation nodes form an inorder tree with respect to their node labels. A special processor called *implicit routing node* is needed to connect the operation nodes on a two-hop edge. Notice that the data nodes are used as operation nodes or implicit routing nodes also. What we have to do is to construct each edge efficiently, including computing the source and destination node labels of each edge and determining the implicit routing node if necessary.

In the complete inorder tree of \( 2^n - 1 \) operation nodes, there are totally \( n \) levels indexed from bottom to top by 0 to \( n - 1 \). In level \( l \), there are \( 2^{n-l-1} \) nodes indexed from left to right, beginning with 0. The label of the root node is \( 2^n - 1 \), and all the labels of leaf nodes are even numbers. Let \( k \) be the label associated with the operation node located at the \( j \)th position of level \( l \). It can be shown that \( k = (2j + 1)2^{l-1} - 1 \) [9]. By induction on the height of the tree, it can be shown that the labels of the parent, left child and right child of node \( k \) are \( 2^{l+1}[k/2^{l+1}] + 2^{l+1} - 1, k - 2^{l-1} \) and \( k + 2^{l-1} \), respectively. To complete the description, we define the label of the parent of the root to be \( -1 \). Given a node \( k \), if we can compute the value of \( 2^i \) efficiently, it will become easy to know the labels of its parent and children.

**Theorem 3.** Let \( P_k \), \( LC_k \) and \( RC_k \) be respectively the labels of the parent, left child and right child of a node \( k \) in a complete inorder tree, and let \( X = k + 1 \). Then

\[
2^l = (X \oplus k) \oplus ((X \oplus k) \gg 1),
\]

\[
P_k = (k + b(X \oplus k) \bullet_h (2^{l+1})) - (k + b(X \oplus k) \bullet_h (1^{n-2}01)) = 4|k/4| + 1,
\]

\[
LC_k = k - (2^l \gg 1), \quad RC_k = k + (2^l \gg 1).
\]

**Proof.** If \( k \) is even, node \( k \) is in level 0 of the inorder tree. It is obvious that \( 2^l = (X \oplus k) \oplus ((X \oplus k) \gg 1) = 1 \), and

\[
P_k = (k + b(X \oplus k) \bullet_h (2^{l+1}))
\]

\[
= (k + b(X \oplus k) \bullet_h (1^{n-2}01)) = 4|k/4| + 1,
\]

\[
LC_k = k, \quad RC_k = k + 1.
\]

If \( k \) is odd and node \( k \) is at the \( j \)th position of level \( l \), then \( k = (j + 1)2^l - 1 \). Since \( X = k + 1 \), we have \( X \oplus k = 0^{n-l-1}1^{l+1} \) and \( 2^l = (X \oplus k) \oplus ((X \oplus k) \gg 1) \).

It is now easy to compute the values of \( P_k \), \( LC_k \) and \( RC_k \). By the inorder property of the tree, \( P_k = 2^{l+1}[k/2^{l+1}] + 2^{l+1} - 1 \). By investigating the binary representation of \( P_k \), we can conclude that it must have the form

\[k_{n-1}k_{n-2} \ldots k_{l+2}011 \ldots 11.\]

So, \( P_k = (k + b(X \oplus k)) \bullet_h (2^{l+1}) \). At last, we can also derive that \( LC_k = k - (2^l \gg 1) \) and \( RC_k = k + (2^l \gg 1) \). □

The following two lemmas are useful for constructing an incomplete inorder spanning tree and for studying the fault tolerance capability of the tree.
Lemma 4. In a complete inorder tree, suppose node \( k \) in level \( 1 \) is reachable from node \( r \) by first traversing along the right edge of node \( r \) and then traversing along subsequent left edges, then \( k = r + 2^l \).

Proof. Let node \( x \) in level \( l \) be the right child of node \( r \). Then \( x = r + 2^l \). Since \( x = k + 2^l + 2^l+1 + \cdots + 2^l+1 = k + 2^{l+1} - 2^l \), we have \( k = r + 2^l \). □

Lemma 5. In a complete inorder tree, suppose node \( k \) in level \( 1 \) is reachable from node \( r \) by first traversing along the left edge of node \( r \) and then traversing along subsequent right edges, then \( k = r - 2^l \).

Proof. Let node \( x \) in level \( l \) be the left child of node \( r \). Then \( x = r - 2^l \). Since \( x = k - 2^l - 2^l+1 - \cdots - 2^l+1 = k - 2^l + 2^l \), we have \( k = r - 2^l \). □

An incomplete inorder tree is an inorder tree whose left subtree is a complete tree and whose right subtree is an incomplete tree. Figure 3 shows an incomplete inorder spanning tree. Let \( N' \) be the size of the tree. We call a node a "break point" if its left and right subtrees are rooted at different levels, like the root node in Fig. 3. There may be many break points in an incomplete inorder tree. We always traverse along the right-edge way to reach the break points.

![Fig. 3. An incomplete inorder spanning tree.](image_url)

Lemma 7. If node \( r \) is a break point in an incomplete inorder tree, then its right child node \( k \) is in level \( l = \lfloor \log(N' - 1 - r) \rfloor \). If \( r + N' - 1 \) then \( k = r + 2^l \); otherwise, \( r \) has no right child.

Proof. If \( r = N' - 1 \), it is the last node which, of course, has no right child. Otherwise, since node \( r \) is a break point, we can imagine to augment the missing edges from node \( r \) to node \( k \) that should appear in a complete inorder tree. These edges consist of the right edge of node \( r \) and subsequent left edges. By Lemma 4, \( k = r + 2^l \). Since \( k \leq N' - 1 \) and \( k \) is the highest node under node \( r \), we have \( l = \lfloor \log(N' - 1 - r) \rfloor \). □

Now consider a data set of arbitrary \( N' \) items which is mapped by the BRGC scheme to contiguous processors with the starting address \( s \). To build an inorder spanning tree over the \( N' \) processors, each processor first independently computes the labels of its neighboring operation nodes using the formulas in Theorem 3. Since the tree may be incomplete, each processor should perform a range checking on the labels computed to ensure the integrity of the tree. When a non-root node \( k \) at level \( l \) finds the label of its parent greater than \( N' - 1 \), by Lemma 4, the label of its parent is set to be \( k - 2^l \). Similarly, if node \( k \) finds its right child out of the range, by Lemma 7, it resets the label of its right child to be \( k + 2^{\lfloor \log(N' - 1 - k) \rfloor} \) if \( k \neq N' - 1 \); otherwise, reset its right child to be \( k \) itself. Note that the inorder spanning tree remains to have dilation 2, because the difference between the two node labels of an edge is still a power of two. The actual processor address corresponding to a node labelled \( m \) is computed by \( G(G^{-1}(s) + m) \).

To build multiple inorder spanning trees is easy. We can map the data sets to consecutive segments of processors and build an inorder spanning tree for each data set using the aforementioned method.

4. Routing in communication trees

Since the constructed inorder spanning tree has a dilation of 2, the choice of implicit routing
nodes is important for avoiding edge conflict in data routing. The following theorem can be used to decide the implicit routing nodes for all edges of dilation 2.

**Theorem 8.** Let \( y = x + 2^i \), \( i > 0 \). There exists a unique number \( z \) such that \( x < z < y \) and both \( G(x) \) and \( G(y) \) differ with \( G(z) \) in one bit. Specifically, if \( x_{i-1} = 0 \) then \( z = x \oplus (2^i - 1) \); otherwise, \( z = y \oplus (2^i - 1) \).

**Proof.** Let \( x_{[a:b]} \) denote the bits \( x_a, x_{a+1}, \ldots, x_b \). Since \( y - x = 2^i \) and \( i > 0 \), there exists a bit \( j \) such that \( j > i \) and

\[
x = x_{n-i}x_{n-2} \ldots x_{j+1}1011 \ldots 1x_{i-1} \ldots x_0,
y = y_{n-i}y_{n-2} \ldots y_{j+1}1000 \ldots 0y_{i-1} \ldots y_0,
\]

where \( x_{[0:i-1]} = y_{[0:i-1]} \) and \( x_{[i+1:n-i]} = y_{[i+1:n-i]} \).

By Lemma 1,

\[
G(x) = G(x)_{n-i}G(x)_{n-2} \ldots G(x)_{j+1}G(x)_j \ldots \ldots 0G(x)_{i-1}G(x)_{i-2} \ldots G(x)_0,
\]

\[
G(y) = G(y)_{n-i}G(y)_{n-2} \ldots G(y)_{j+1}G(y)_j \ldots \ldots 0G(y)_{i-1}G(y)_{i-2} \ldots G(y)_0,
\]

where \( G(x)_{[0:i-2]} = G(y)_{[0:i-2]} \) and \( G(x)_{[i+1:n-i]} = G(y)_{[i+1:n-i]} \).

Since \( G(x) \) and \( G(y) \) differ in two bits, there are two numbers that differ with both \( G(x) \) and \( G(y) \) in one bit. Let \( G(s) \) and \( G(t) \) be the two numbers, then

\[
G(s) = G(x)_{n-i}G(x)_{n-2} \ldots G(x)_{j+1}G(x)_j \ldots \ldots 0G(x)_{i-1}G(x)_{i-2} \ldots G(x)_0,
\]

\[
G(t) = G(x)_{n-i}G(x)_{n-2} \ldots G(x)_{j+1}G(x)_j \ldots \ldots 0G(x)_{i-1}G(x)_{i-2} \ldots G(x)_0.
\]

By Lemma 1,

\[
s = x_{n-i}x_{n-2} \ldots x_{j+1}0111 \ldots 1x_{i-1}x_{i-2} \ldots x_0,
\]

\[
t = x_{n-i}x_{n-2} \ldots x_{j+1}1000 \ldots 0x_{i-1}x_{i-2} \ldots x_0.
\]

Therefore, if \( x_{i-1} = 0 \) then \( s < x < y \) and \( x < y < t \). If \( x_{i-1} = 1 \) then \( s < x < y \) and \( x < t < y \). In both cases, there exists a unique number \( z \) between \( x \) and \( y \) such that \( G(z) \) differs with both \( G(x) \) and \( G(y) \) in one bit. Specifically, if \( x_{i-1} = 0 \) then \( z = x \oplus (2^i - 1) \); otherwise, \( z = y \oplus (2^i - 1) \). □

For a node \( x \) to communicate with its neighboring node \( y \) in the spanning tree, if nodes \( x \) and \( y \) are one hop away, no implicit routing node is needed. Otherwise, node \( x \) computes the implicit routing node as described in Theorem 8. The routing rule ensures the communication integrity of the spanning trees. That is, their communications do not interfere with each other. In the following, we analyze the node and edge congestion for the normal tree algorithms.

**Corollary 9.** All implicit routing nodes have even labels.

**Proof.** Let node \( z \) be the implicit routing node of an edge which is of dilation 2 and connects two operation nodes \( x \) and \( y \). Then \( x \) and \( y \) are both odd and their difference is a power of two. By Theorem 8, both \( (z \oplus x)_0 \) and \( (z \oplus y)_0 \) are 1. Since both \( x \) and \( y \) are odd, \( z \) must be even. □

**Corollary 10.** An implicit routing node \( r \) can only appear in the unique path from the root to the leaf operation node \( r \) in the inorder tree.

**Proof.** By Corollary 9, node \( r \) is a leaf operation node of the inorder spanning tree. By Theorem 8, the implicit routing node \( r \) may occur only in those edges connecting two operation nodes whose labels encompass \( r \). In an inorder tree, the edges that encompass \( r \) are those comprising the binary search path for \( r \) in the tree. The searching path is just the path from the root to the leaf operation node \( r \). □

**Corollary 11.** Let \( s \) be the starting address of a processor segment. Suppose \( r \) is an implicit routing node and \( x \) is a non-leaf operation node. Then the communication channel connecting processors \( G(G^{-1}(s)+x) \) and \( G(G^{-1}(s)+r) \) may be used only in the edge connecting \( x \) and its parent and the edge connecting \( x \) and one of its children.
Proof. Node \( x \) has only three edges incident on it, one to its parent, and two to its children. By Corollary 10, it is impossible that an implicit routing node \( r \) occurs in both \( x \)'s child edges. However, node \( r \) may be encompassed by node \( x \) and its parent node. So, the communication channel connecting processors \( G(G^{-1}(s) + x) \) and \( G(G^{-1}(s) + r) \) may be used by at most two edges in the inorder tree, one connecting \( x \) to its parent and the other connecting \( x \) to one of its children. \( \Box \)

Since the segment spanning trees act without interference to each other, analysis of a single tree is enough. In a normal tree algorithm, only one level of processors is active at a time. Due to Corollary 11, no node or edge congestion is incurred.

5. Constructing large trees and fault tolerance

Suppose there are \( t \) inorder spanning trees of sizes \( N_1, N_2, \ldots, N_t \) to be constructed on a hypercube with \( P \) processors, where \( P < N_1 + N_2 + \cdots + N_t \). We present a very straightforward allocation scheme in which the load of each processor is averaged. First allocate to each spanning tree a number of physically contiguous processors proportional to the tree size. Then, each tree stores its data items in a consecutive storage scheme to the processors allocated to it, and an inorder spanning tree is constructed on the processors using the method described previously. Figure 4 illustrates the construction of two trees \( A \) and \( B \) of sizes 8 and 24, respectively, on a hypercube with 8 processors.

As concerning the fault tolerance of the inorder spanning trees, we first consider the fault tolerance capability against the leaf (operation node) failures. If only one leaf descendent fails, the parent will perform the task of the faulty leaf. If both leaf descendents fail, the two nodes labelled next to the corresponding faulty leaves will replace them. Since the two replacement nodes differ by 2, they are still two hops away. With this leaf replacement scheme, the trees can tolerate a small fraction of leaf failures with a little slowdown, which is due to the missing of some implicit routing nodes.

Assuming no leaf failures, the embedding method provides effective fault tolerance against internal node failures. Suppose \( x \) is a faulty internal node. If \( x \) is a left descendent, the right child of \( x \) is used to replace \( x \). On the other hand, if \( x \) is a right descendent, \( x \) can be replaced with its left child.

![Diagram of two trees](image-url)

Fig. 4. Build two large trees on a 3-cube. (Tree \( A \) uses two processors and tree \( B \) uses six processors.)
Theorem 12. The replacement scheme for internal nodes preserves the dilation of the embedding at 2.

Proof. Let node $r$ be the parent of the faulty node $x$ which is at level $l$ and node $k$ be the selected node for replacement. If node $x$ is a left child of node $r$ (Fig. 5(a)) by Lemma 5, node $k$ and node $r$ are two hops away. Similarly, if node $x$ is a right child of node $r$ (Fig. 5(b)), by Lemma 4, node $k$ and node $r$ are two hops away. In both cases, node $k$ is two hops away from the other child node $t$ of $x$, since the difference of their labels is $2^l$. □

6. Concluding remarks

We have presented an efficient, actually constant-time, construction of segment inorder spanning trees on consecutive data sets in a hypercube. This data structure is well suited for segmented scan and image processing problems in which data are mapped to contiguous processors for convenient neighborhood communication. It is also shown that our routing strategy is edge congestion free for normal tree algorithms. The issue of constructing large trees on a small hypercube is solved straightforwardly. Replacement scheme for fault tolerance is presented. In addition, since the tree sizes can be arbitrary, our results can be extended to incomplete hypercube easily. Position flexibility is another advantage of the inorder trees, so we can devise a relocatable distributed data structure for fault tolerant parallel computing.

References