LINEARLY ORDERED CONCURRENT DATA STRUCTURES ON HYPERCUBES

THESIS

Presented to the Graduate Council of the University of North Texas in Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Ajita John, B.Sc., B.E.
Denton, Texas
August, 1992
This thesis presents a simple method for the concurrent manipulation of linearly ordered data structures on hypercubes. The method is based on the existence of a pruned binomial search tree rooted at any arbitrary node of the binary hypercube. The tree spans any arbitrary sequence of $n$ consecutive nodes containing the root, using a fan-out of at most $\lceil \log_2 n \rceil$ and a depth of $\lceil \log_2 n \rceil + 1$. Search trees spanning non-overlapping processor lists are formed using only local information, and can be used concurrently without contention problems. Thus, they can be used for performing broadcast and merge operations simultaneously on sets with non-uniform sizes. Extensions to generalized and faulty hypercubes and applications to image processing algorithms and for $m$-way search are discussed.
ACKNOWLEDGEMENTS

I owe a lot to my advisor Dr. Sajal Das; he provided constant advice and technical guidance. I am expressing here my sincere gratitude to him. I also thank Dr. Joydeep Ghosh at the University of Texas at Austin; he initiated this work and has constantly provided valuable suggestions.

I thank Dr. Tom Jacob and Dr. Ian Parberry for their willingness to be members of my supervising committee. I am indebted to Dr. Fisher for his guidance and financial support. I thank the department for the computing facilities offered to me and all the members of the office staff who have always helped me with any paperwork.

Finally, I am grateful to my family and friends for their love and support.

Ajita John

The University of North Texas

August, 1992
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ................................................................. iii
LIST OF TABLES ........................................................................ vi
LIST OF ILLUSTRATIONS ............................................................. vii
1. INTRODUCTION ..................................................................... 1
2. MAPPING LINEAR ORDER ONTO A HYPERCUBE ....................... 3
   2.1 Binary Hypercubes ........................................................... 3
   2.1 Mapping a Linear Order ................................................... 4
3. SPANNING AND PRUNED BINOMIAL TREES ............................ 7
   3.1 Spanning Binomial Trees .................................................. 7
   3.1.1 Communication Types ................................................. 8
   3.2 Pruned Binomial Trees ..................................................... 10
   3.2.1 Characterization of Spannable Intervals ....................... 11
4. SEARCH TREES AND PROPERTIES OF PBTs ......................... 17
   4.1 Construction of Search Trees .......................................... 17
   4.2 Invariance of PBTs .......................................................... 20
5. EXTENSIONS OF PBTs............................................................. 23
   5.1 Processing of Multidimensional Data ................................ 23
   5.2 Coarser Granularity .......................................................... 24
LIST OF TABLES

3.1 Spanning Intervals (SIs) for different line segments. .............. 15
LIST OF ILLUSTRATIONS

2.1 Gray code mapping on a 3-dimensional hypercube. .................. 5
3.1 Examples of n-level binomial trees. ................................. 8
3.2 SBT in a 3-cube. ................................................. 9
3.3 Broadcasting in a 3-cube. ......................................... 10
3.4 A PBT for interval [8, 15]. ..................................... 12
3.5 A PBT that does not span an interval. ............................ 12
3.6 A Type II PBT for interval [20, 27]. .............................. 14
3.7 A maximal SI for [18, 28]; ..................................... 16
3.8 A maximal SI for [18, 26]. ..................................... 16
4.1 Search tree for interval of [18,28] with R = 25; ........................ 20
4.2 Search tree for interval of [18,26] with R = 26. .................. 21
4.3 Translation may not confine a search tree to an interval. ...... 22
5.1 6-ary 3-cube .................................................. 33
5.2 8-ary 2-cube .................................................. 33
5.3 Generalized Gray code mapping on a 4-ary 2-cube ............... 35
5.4 PKT in 4-ary 3-cube for interval [16, 31] .......................... 37
6.1 An h-level graph; .............................................. 46
6.2 A PBT for illustrating 8-way search. ................................ 47
6.3 Mapping of a $4 \times 4$ matrix onto the hypercube nodes. . . . . . . . . . . . . . . . . . . 49
CHAPTER 1

INTRODUCTION

Concurrent data structures facilitate the development and execution of parallel programs by providing built-in mechanisms for communication and synchronization [12, 20]. In a message-passing multicomputer such as the hypercube, the data elements are distributed over the processor/memory nodes, and computation is performed by all the processors where these elements reside. In general, operations over distributed data also require interprocessor communication and coordination for correctness and consistency of the concurrent operations.

The parallelism afforded by concurrent data structures on hypercube multicomputers has been investigated by several researchers. Dally [12] introduced the concept of balanced cube as a concurrent data structure for ordered sets and showed how $O\left(\frac{N}{\log N}\right)$ concurrency could be obtained for searching among $N$ elements. Mu and Chen [28] focused on broadcast and merge operations carried out on multiple sets in parallel with complexities independent of the number of sets involved. A two-level scheme for dictionary operations with medium granularity has been recently proposed by Dobes et. al. [15]. Ho and Johnsson [23] have developed algorithms to support broadcasting for uniform sized static sets distributed over hypercubes. Leiserson and Maggs [27] have considered communication issues over dynamic sets and developed randomized algorithms for the fat-tree machine model.

All these approaches involve the formation of communication-efficient broadcast/search trees among the processor nodes. Consequently, there has been considerable work on the efficient embedding of tree structures in hypercubes [23, 43, 44, 45]. However, these methods apply to complete subcubes rather than an arbitrary subset.
of the processors, and do not cater to irregular problems. A comprehensive review of mapping of regular problems on hypercubes is given in [18].

It is well known that proximity preserving binary-reflected Gray code can be used to find a Hamiltonian cycle (and hence to embed a linear chain) in the binary hypercube [18, 24, 35]. Recently, a strong case has been made for considering sets and arrays as basic data structures in parallel programming [11, 38]. As shown in Chapter 6, the concurrent manipulation of static sets can be formulated as operations on disjoint segments of a linear chain embedded in the hypercube. Indeed, the ability for concurrent search within segments of such embeddings has been recognized as crucial for distributed set manipulation [28], in which an algorithm is presented for generating search trees with depth $2 \log n$ for a segment of size $n$.

Ghosh and Deshpande [20] proposed an alternate scheme that yields broadcast/search trees, with about half that depth, in a simple fashion. This thesis is a refinement and continuation of the work initiated by them. The trees have a perfect embedding with both expansion and dilation of one, and span a segment of arbitrary length starting from an arbitrary element within that segment. This thesis further investigates the properties of these trees and presents several extensions and applications. Chapter 2 introduces how to map a linear ordering onto a hypercube in terms of Gray codes. Chapter 3 investigates the nature of segments spanned by a variant of binomial trees, called pruned binomial trees, built on top of a linear data structure. The construction of the "search" trees is detailed in Chapter 4. Extensions to multidimensional data, coarser granularities, faulty hypercubes and generalized hypercubes, are given in Chapter 5, while several applications of the concurrent data structure are outlined in Chapter 6. Chapter 7 offers conclusions.
CHAPTER 2

MAPPING LINEAR ORDER ONTO A HYPERCUBE

2.1 Binary Hypercubes

A $n$-dimensional binary hypercube (also called binary $n$-cube) is a multi-computer network with $N = 2^n$ nodes and diameter $n$, where the diameter is the maximum number of edges in the shortest path between any pair of nodes. There are $\binom{2^n}{2}$ nodes at Hamming distance\(^1\) of $x$ from a given node, and $n$ node-disjoint paths between any pair of nodes $i$ and $j$. The fanout (i.e. degree) of every node is $n$, and the total number of communication links is $\frac{1}{2}N\log N$ (Unless otherwise specified, all logarithms are to the base two).

Each node of the $n$-dimensional hypercube can be specified by the binary address $(b_{n-1}, b_{n-2}, ..., b_0)$, where $b_i$ is a binary digit. The $m$th bit corresponds to the $m$th dimension in a Boolean space. The dimensions are specified by the set $\{0, 1, ..., n - 1\}$. Two nodes with addresses $(b^1_{n-1}, b^1_{n-2}, ..., b^1_0)$ and $(b^2_{n-1}, b^2_{n-2}, ..., b^2_0)$ are connected by an edge (or link) in the $m$th dimension if and only if

$$b^1_m = b^2_m \quad \text{and} \quad b^1_i = \overline{b}^2_i, \quad \text{for } 0 \leq i \leq n - 1, i \neq m$$

where $\overline{b}^2_m$ is the complement of $b^2_m$. A 3-dimensional binary hypercube is shown in Figure 2.1. The node addresses are given in the parentheses.

A path between two nodes with addresses $(b^1_{n-1}, b^1_{n-2}, ..., b^1_0)$ and $(b^2_{n-1}, b^2_{n-2}, ..., b^2_0)$, is specified by a sequence of edges. The number of edges in this

\(^1\)The Hamming distance between a pair of binary numbers, or nodes, $i$ and $j$ is $\text{Hamming}(i, j) = \sum_{m=0}^{n-1} (i_m \oplus j_m)$.\
path is the number of bits in which the addresses of the two nodes differ; and each edge corresponds to a bit in the address, \((b_{n-1}^1, b_{n-2}^1, \ldots, b_0^1)\) being corrected to the corresponding bit in address \((b_{n-1}^2, b_{n-2}^2, \ldots, b_0^2)\). A \(j\)-subcube of a binary hypercube is a subgraph consisting of \(2^j\) nodes, with the connecting links between them, obtained by choosing any \(n - j\) dimensions and considering all the nodes that have the same address bit in each of these dimensions.

2.2 Mapping a Linear Order

In several applications it is useful to impose a linear order on the data elements. For example, one can order the entries in a dictionary alphabetically so as to facilitate entry location, insertion or deletion. For a data structure with \(N\) linearly (i.e., strictly as opposed to partially) ordered elements, one can assign a unique index \(0 \leq I \leq N - 1\) to each element that specifies its position in the linear order.

Consider the mapping of these elements onto an \(n\)-dimensional binary hypercube, using the binary reflected Gray code [24, 30]. For the moment, we assume that exactly one datum is mapped onto each processor such that \(N = 2^n\). The case where \(N > 2^n\) and the accompanying issues of machine granularity will be examined in Section 5.2. A Gray code embedding specifies that if \(I = \sum_{i=0}^{n-1} b_i 2^i\), then the element with index \(I\) is mapped onto a hypercube node with address \(G(I) = g_{n-1} \ldots g_0\), where:

\[
g_i = \begin{cases} 
  b_i \oplus b_{i+1}, & \text{if } i < n - 1 \\
  b_i, & \text{if } i = n - 1.
\end{cases}
\] (2.1)

In other words, \(G(I)\) is a bit vector generated by taking the modulo-2 sum or the Exclusive-OR operation, denoted by \(\oplus\), of adjacent bits of the binary encoding of index \(I\).

Example 1: Consider the linear ordering \(a_0, a_1, \ldots, a_7\) of eight elements. The 3-bit binary encoding of the indices of these elements are 000, 001, 010, 011, 100, 101,
Figure 2.1: Gray code mapping on a 3-dimensional hypercube.

110, and 111. The corresponding Gray code encodings (shown in parentheses) are respectively 000, 001, 011, 010, 110, 111, 101, and 100. The processors onto which the given elements are mapped in a three-dimensional hypercube are shown in Figure 2.1.

It can be seen from the thick lines in Figure 2.1 that the Gray code embedding ensures that adjacent elements in the linear ordering are mapped onto adjacent processors. Moreover, this code has useful properties of symmetry and reflection brought out in the following alternate, recursive definition [12, 35]:

\[
\text{GLIST}(1) = [G(0), G(1)] = [0, 1];
\]

\[
\text{GLIST}(n + 1) = [0G(0), 0G(1), \ldots, 0G(2^n - 1), 1G(2^n - 1), \ldots, 1G(0)],
\]

where \(\text{GLIST}(n) = [G(0), G(1), \ldots, G(n)]\) is the list representation of the \(n\)-bit codes of the first \(2^n\) integers.

A fundamental relation between linearly ordered indices that are mapped onto adjacent hypercube nodes, is established by the following theorem [20] which will be used frequently in this thesis.

**Theorem 1** Let \(R\) and \(S\) be linear indices of two elements. If a Gray code embedding
results in mapping these two elements onto hypercube nodes that are neighbors in the $k^{th}$ dimension, for $0 \leq k \leq n - 1$, then the binary representation of $S$ can be obtained by complementing the $k + 1$ least significant bits of the binary representation of $R$.

We give the proof of Theorem 1 below.

Proof: By interchanging $g_i$ and $b_i$ in Eq. (1), we get

$$b_i = \begin{cases} g_i + b_{i+1} & \text{if } i < n - 1 \\ g_i & \text{if } i = n - 1. \end{cases} \quad (2.3)$$

Let indices $R$ and $S$ be mapped onto nodes $G(R) = g_{n-1}' \ldots g_0'$ and $G(S) = g_{n-1}' \ldots g_0'$ respectively. Since $G(R)$ and $G(S)$ differ only in bit $g_k$, it follows from Eq. (2.3) that

$$b'_i = b'_i, \quad \text{for } k < i \leq n - 1;$$

$$b'_k = g_k' + b'_{k+1} = g_k' + b'_{k+1} = (g_k' + 1) + b'_{k+1}$$

$$= 1 + b_k' = b_k'.$$

Substituting back in Eq. (2.3) for $i = k - 1, k - 2, \ldots, 0$, we get $b'_i = b'_i$, for $0 \leq i \leq k$. 

The following two corollaries are immediate from the preceding theorem.

**Corollary 1** By traversing a finite sequence of edges in dimensions $0$ through $k$, it is possible to reach any element that differs from $R$ in the $k + 1$ least significant bits, but no element that differs in the $(k + 2)^{th}$ or more significant bits.

**Corollary 2** The expression $b_j \oplus b_{j+1}$ has the same value for the addresses of all nodes visited by traversing a sequence of edges in dimensions other than $j$. 
CHAPTER 3

SPANNING AND PRUNED BINOMIAL TREES

3.1 Spanning Binomial Trees (SBTs)

Definition 1: A binomial tree is defined recursively as follows [2]:

(i) A zero-level binomial tree has a single node.

(ii) An $n$-level binomial tree is constructed out of two $(n-1)$-level binomial trees by adding an edge between the roots of the two trees, and then making either root the new root.

Binomial trees for levels 0-3 are shown in Figure 3.1.

It can be shown that an $n$-level binomial tree has $\binom{n}{i}$ nodes at level $i$, and it is composed of $n$ subtrees, each of which is a binomial tree of level 0, 1, ..., $n - 1$ respectively.

An $n$-level binomial tree can be embedded in a binary $n$-cube (consisting of $2^n$ nodes) as a spanning tree, and is therefore also known as a spanning binomial tree (SBT) according to Ho and Johnsson [23]. The spanning tree rooted at node 0 of a hypercube contains the edges that connect a node $i$ with the subset of its neighbors having addresses generated by complementing any bit of leading zeros of the binary encoding of $i$. For an arbitrary source node, $s$, the spanning tree is simply translated by a bit-wise Exclusive-OR operation on all addresses with the address of the source node. The spanning binomial tree rooted at node $s$ and denoted as $T^{SBT}(s)$ can be defined as follows. Let $i$ be any arbitrary node and $i \oplus s = c = (c_{n-1}c_{n-2}...c_0)$, where $c_m = i_m \oplus s_m$. Let $p$ be such that $c_p = 1$ and $c_m = 0, \forall m \in \{p+1, p+2, ... , n-1\} \equiv M^{SBT}(c)$ and let $p = -1$ if $c = 0$. The set $M^{SBT}(c)$ is the set of leading zeros of $c$. Then,
Figure 3.1: Examples of n-level binomial trees.

\[
\text{children}^{SBT}(i, s) = \{(i_{n-1}i_{n-2}\ldots i_m\ldots i_0)\}, \forall m \in M^{SBT}(c),
\]

\[
\text{parent}^{SBT}(i, s) = \begin{cases} 
\phi & i = s \\
(i_{n-1}i_{n-2}\ldots i_p\ldots i_0), & i \neq s.
\end{cases}
\]

It can be seen that node \( j \) is a child of node \( i \) iff node \( i \) is the parent of node \( j \).

3.1.1 Communication Types

Two types of communication in a hypercube are \textit{broadcasting} and \textit{personalized communication} [23]. In broadcasting, a data set is copied from one node to all other nodes, or a subset thereof. In personalized communication, a node sends a unique data set to all other nodes, or a subset thereof. Thus, unlike broadcasting, no replication/reduction of data takes place in personalized communication. In one-port communication, a processor can only send and receive on one of its ports at any time. The port on which a processor sends and receives data can be different. In n-port communication, a processor can communicate on all ports concurrently. For broadcasting, at node \( i \), the data are replicated as many times as the number of children of \( i \). With n-port communication, all ports are scheduled concurrently. With one-port communication, the order of communications on different ports is important. In
personalized communication, the source node sends a unique message to every other node. An internal node needs to receive and forward all the data for every node of the subtree of which it is a root. The ordering of data for a port is important for the communication time - both for one-port and n-port communication.

Any spanning tree can be used to broadcast data from a single source to all other nodes. In particular, a Hamiltonian path is also a spanning tree. A node replicates the data as many times as the out-degree of the node in the spanning tree. In broadcasting one element the minimum number of routing steps is \( \log N \). Any spanning tree can achieve this lower bound, if each node can send out data through all the links connected to it during one step, i.e. n-port communication. In case each node can send or receive data through only one link during one step, i.e. one-port communication, then only the class of spanning binomial trees can attain the lower bound, since after each broadcasting step the number of nodes that own the desired data is at most twice that of the previous step. This is known as the binomial broadcasting [23]. Figure 3.2 shows the SBT for a binary 3-cube. Figure 3.3 shows the broadcasting in a 3-cube for the SBT in Figure 3.2. Labels on edges are time steps of broadcasting for one-port communication.
3.2 Pruned Binomial Trees (PBTs)

Ghosh and Deshpande [20] generalized the concept of an SBT to include any binomial tree that is formed by traversing only edges corresponding to selected dimensions of a hypercube. These pruned binomial trees (PBTs) span a subspace of the hypercube defined by fixing the values of the bits only for those dimensions that are not traversed, and allowing all possible combinations of values for the other bits. Thus PBTs are structurally similar to the SBTs. Indeed, the analysis on broadcasting, personalized communication, communication using a single port at a time, discussed in [23, 25] all carry over. However since a PBT can be built on top of a linear embedding, it has added useful properties as brought out in the following sections.

The dimensions that can be traversed by a given PBT are referred to as the flip dimensions for that tree. These dimensions form the flip-set, \( \mathcal{F} \). Thus a PBT is completely defined by the root address \( R \) and the set \( \mathcal{F} \), and can be generated by the following algorithm.
Algorithm 1: PBT-GENERATE(R, F)

(1) send (∞, F) to root processor G(R) (* initialization *)

(2) for all processors
   receive(dimension, flip-set);
   for all D ∈ flip-set in parallel do
      send (D, {e : (e ∈ flip-set) ∧ (e < D)} ) to neighbor in Dth dimension
      (* claim neighbor as child *)
   enddo.

Definition 2: An interval [c, d] in a linear ordering is the set of elements with consecutive indices c, c + 1, ..., d. The length of this interval is \( L(c, d) = d - c + 1 \).

Example 2: Figure 3.4 shows a PBT rooted at node address (1010) and spanning elements in the interval [8, 15]. Here the flip-set is \( F = \{0, 1, 2\} \), and the hypercube node addresses (i.e., Gray codes) are enclosed in parentheses. Note that each tree link corresponds to a direct connection between two adjacent hypercube nodes. In this case, the PBT is the same as a three-level SBT in Figure 3.2, and spans the physical subcube 1xxx. Figure 3.5 shows a PBT rooted at (0101), with flip dimensions in \( F = \{0, 2, 3\} \). This tree spans elements with indices 0, 1, 6, 7, 8, 9, 14, 15 which do not form an interval.

3.2.1 Characterization of Spannable Intervals

Let us now characterize those PBTs that span an interval, and obtain the interval corresponding to a given root address and flip dimensions.

Definition 3: A spannable interval (SI), denoted as \([a, b]\), is a set of elements with consecutive indices \( a, a + 1, \ldots, b \) such that, for every element \( α \in SI \), there exists a PBT rooted at \( α \) that spans exactly those elements that constitute the set.
Figure 3.4: A PBT for interval [8, 15].

Figure 3.5: A PBT that does not span an interval.
We present a modification of a corresponding theorem in [20].

**Theorem 2** When the Gray code is used to embed a linear chain in a hypercube of the same size, an interval \([a, b]\) in this chain is an SI iff there exist non-negative integers \(p, j\) and \(k\) such that \(p\) is odd, \(j > k\) and

\[
a = p2^j - 2^{k+1}, \quad b = p2^j + 2^{k+1} - 1.
\]

(\(\Rightarrow\)) Given below are two types of PBTs that span intervals conforming to Eq. (3.1).

**Type I:** Select a root \(R\) such that \(p = b_{n-1}^r \ldots b_j^r\), given by the \(n - j\) most significant bits of the binary representation of \(R\). Also, let \(\mathcal{F} = \{0, 1, \ldots, j\}\). Since \(p\) is odd, \(b_j^r = 1\). The resultant PBT spans the logical subspace \(b_{n-1}^r \ldots b_{j+1}^r x \ldots x\), covering the interval \([(p - 1)2^j, (p + 1)2^j - 1]\), which conforms to Eq. (3.1) for \(j = k + 1\). The PBT of Figure 3.4 spans an SI of Type I.

**Type II:** Select a root \(R\) such that \(b_j^r b_{j-1}^r \ldots b_{k+1}^r = 01 \ldots 1\) or \(10 \ldots 0\); and let \(\mathcal{F} = \{0, 1, \ldots, k < j - 1; j\}\). A PBT of this type will span an interval given by

\[
[b_{n-1}^r \ldots b_{j+1}^r 01 \ldots 1 b_k^r 0 \ldots 0, b_{n-1}^r \ldots b_{j+1}^r 10 \ldots 0 b_k^r 1 \ldots 1],
\]

where \(b_k^r = 0\) for the lower bound value and \(b_k^r = 1\) for the upper bound value. Then \(a\) and \(b\) are given by Eq. (3.1). Such a PBT is shown in Figure 3.6, which spans the interval \([20, 27]\) such that \(\mathcal{F} = \{0, 1, 3\}\).

(\(\Leftarrow\)) If a PBT does not belong to any of the above two categories, then there exist dimensions \(u, v, w\) and \(z\) such that (i) \(v = u - 1 > w\), (ii) \(z = w - 1\), (iii) \(u\) and \(w\) are flip dimensions, and (iv) \(v\) and \(z\) are not flip dimensions. In such cases, Corollary 2 states that both \(b_u \oplus b_v\) and \(b_w \oplus b_z\) are constant. For the spanned elements to be consecutive, we require \(b_u \oplus b_v = 1\) since otherwise some numbers with \(b_u b_{u-1} = 00\) or \(11\) are spanned while none of the numbers with \(b_u b_{u-1} = 01\) or \(10\) are covered. Similarly, we require \(b_w \oplus b_z = 1\). However, this means that it is
not possible to reach an element with $b_u b_{u-1} \ldots b_w b_{w-1} = 01 \ldots 11$, even though this falls between reachable elements $b_u b_{u-1} \ldots b_w b_{w-1} = 01 \ldots 10$ and $10 \ldots 01$. Thus the spanned elements are not consecutive.

\begin{quote}
Corollary 3 All intervals $[n 2^j, (n+1)2^j - 1]$ for $n \geq 0$, are SIs of Type I and have length $2^i$.

Having seen what intervals can be considered as SIs, the next task is to characterize the largest SI contained in an arbitrary interval $[c, d]$, of length $L(c, d) = d - c + 1$.

Theorem 3 Any interval $[c, d]$ with length $2^i - 1 \leq L(c, d) < 2^{i+1} - 1$ contains an SI of Type I having length $2^{i-1}$ or more.

Proof: Assuming $n = (c \text{ div } 2^j) + 1$ in Corollary 3, we see that $[n 2^j, (n+1)2^j - 1] \in [c, d]$ is SI of Type I with length $2^j$. Hence the theorem.
Table 3.1: Spanning Intervals (SIs) for different line segments.

Table 1 provides an enumerative version of Theorem 3. It is based on the observation that if $2^l \leq L(c, d) < 2^{l+1}$, then either

(1) $b_n^{d_1} \ldots b_{l+1}^{d_1} = b_{n-1}^{d_1} \ldots b_1^{d_1}; b_i^{d_1} = 1, b_i = 0$; and $b_{l-1}^{d_1} \geq b_{l-1}^{d_1}$, or

(2) $b_n^{d_1} \ldots b_{n-k}^{d_1} = b_{n-1}^{d_1} \ldots b_k^{d_1}$ for some $k > l + 1; b_{n-1}^{d_1} \ldots b_i^{d_1} = 10 \ldots 0; b_{k-1}^{d_1} \ldots b_i^{d_1} = 01 \ldots 1; and b_{l-1}^{d_1} \geq b_{l-1}^{d_1}$.

In Table 1, the entries 1.x give SIs $[a', b']$ for the first case, while the entries 2.x enumerate the possible situations for the second case. These intervals are unique and maximal unless an index $c$ or $d$ happens to be an end-point. Therefore, the maximal SI is given by $[a, b]$, where

$$a = \begin{cases} c & \text{if } b_{l-2}^{d_1} \ldots b_0^{c} = 0 \ldots 0 \\ a' & \text{otherwise} \end{cases} \quad (3.2)$$

and

$$b = \begin{cases} d & \text{if } b_{l-2}^{d_1} \ldots b_0^{d} = 1 \ldots 1 \\ b' & \text{otherwise.} \end{cases} \quad (3.3)$$

Example 3: Figure 3.7 shows an SI of length $2^l = 8$ for the interval $[18, 28]$. This SI can be spanned by a PBT of Type II. Figure 3.8 shows an SI of length $2^{l-1} = 4$ for the interval $[18, 26]$, which can be spanned by a PBT of Type I.
Figure 3.7: A maximal SI for [18, 28];

Figure 3.8: A maximal SI for [18, 26].
CHAPTER 4

SEARCH TREES AND PROPERTIES OF PBTs

4.1 Construction of Search Trees

Given a set of disjoint intervals in our linear ordering, it is possible to obtain a collection of search/broadcast trees – one for each interval – that can be created concurrently using information local to the corresponding intervals.

Let \([a, b]\) be a maximal SI contained in interval \([c, d]\). It can be shown that any element in \([c, d]\) that is not contained in \([a, b]\) can be reached from some element in \([a, b]\) by a single hop [20].

**Lemma 1**

(i) Any element in \([c, a)\) can be reached from some element in \([a, b]\) by traversing dimension \(y_0 = \min \{k \mid k \geq l - 1, b_k^* = 0\}\), and

(ii) Any element in \((b, d]\) can be reached from some element in \([a, b]\) by traversing dimension \(y_1 = \min \{k \mid k \geq l - 1, b_k^* = 1\}\).

**Proof:** By inspection of all the cases enumerated in Table 1 in Chapter 3.

This lemma yields a way to obtain a search tree for \([c, d]\), starting from an arbitrary root within it. The root element first computes \(y_0, y_1, a\) and \(b\) from \(c, d\). Then, if the root element belongs to \([a, b]\), the search tree is the PBT for SI augmented with extra probe links in dimensions \(y_0\) and \(y_1\). If the root is outside \([a, b]\), then we first reach an element within \([a, b]\) in one hop, make this the new "root", and proceed as before. The search tree creation is formalized by the following algorithm.
Algorithm 2: Search-Tree-Generate(R, c, d)

(1) for rootprocessor G(R)

determine \( a, b, y_0, y_1, \mathcal{F}; \)

cobegin

if \( R \in [a, b] \) then

for all \( D \in \mathcal{F} \) in parallel do

send \((D, \{ e : (e \in \mathcal{F}) \land (e < D) \}, 0)\) to neighbor in \( D^{th} \) dimension
enddo;

checksend \((y_0, \emptyset, 1)\) to neighbor in \( y_0^{th} \) dimension;

checksend \((y_1, \emptyset, 1)\) to neighbor in \( y_1^{th} \) dimension;
endif;

if \( R < a \) then send \((y_0, \mathcal{F}, 1)\) to neighbor in \( y_0^{th} \) dimension endif;

if \( R > b \) then send \((y_1, \mathcal{F}, 1)\) to neighbor in \( y_1^{th} \) dimension endif;

coen.

(2) for all other processors

calculate \( y_0, y_1; \)

receive \((dimension, flip-set, flag);\)

cobegin

if flag = 0 then

for all \( D \in \text{flip-set} \) in parallel do

send \((D, \{ e : (e \in \text{flip-set} ) \land (e < D) \}, 0)\) to neighbor in \( D^{th} \) dimension
endo;

checksend \((y_0, \emptyset, 1)\) to neighbor in \( y_0^{th} \) dimension;

checksend \((y_1, \emptyset, 1)\) to neighbor in \( y_1^{th} \) dimension;
endif;

if flag = 1 then

for all \( D \in \text{flip-set, in parallel do} \)

send \((D, \{ e : (e \in \text{flip-set} ) \land (e < D) \}, 0)\) to neighbor in \( D^{th} \) dimension
enddo;
if $y_0 \not\equiv$ dimension, then checksend ($y_0, \{\phi\}, 1$) to neighbor in $y_0^{th}$ dimension;
if $y_1 \not\equiv$ dimension, then checksend ($y_1, \{\phi\}, 1$) to neighbor in $y_1^{th}$ dimension;
endif;
coend.

In Algorithm 2, the index of the neighbor is checked to be in $[c, d]$ before a message is sent to it. It can be shown that if $L(a, b) = 2^l$ then $y_0 = y_1$. Also, note that $\mathcal{F}$ consists of dimensions 0 to $l - 2$. In addition, if $b_{l-1}^d > b_{l-1}^c$, the set $\mathcal{F}$ contains the most significant dimension $k$ in which $b_k^d$ and $b_k^c$ differ.

**Theorem 4** The search tree generated by Algorithm 2 has a maximum depth of $\lceil \log_2 L \rceil + 1$, and a maximum fanout of $\lceil \log_2 L \rceil$, where $L$ is the length of interval $[c, d]$.

**Proof:** A PBT of size $2^l$ has a depth of $l$ and a fanout of $l$. The search trees generated by Algorithm 2 are PBTs, possibly augmented by links in dimensions $y_0$ and $y_1$. Theorem 3 guarantees that one can find a PBT that spans an interval $[a, b]$ of length at least $2^{l-1}$, where $l = \lceil (\log_2 L + 1) \rceil$. We have two cases:

If $L(a, b) = 2^l$, it can be shown that the two elements reached by traversing dimensions $y_0$ and $y_1$ respectively differ by at least $2^{l+1}$. Thus only one of them can belong to $[c, d]$. Thus the maximum fanout is $l + 1 \leq \lceil \log_2 L \rceil$. If $R \in [a, b]$ then a depth of $l + 1$ occurs if some leaf has an augment link. If $R$ is not in $[a, b]$, an extra link is required to reach a new "root" within $[a, b]$, so that maximum the depth is $l + 2 \leq \lceil \log_2 L \rceil + 1$.

If $L(a, b) = 2^{l-1}$, it is possible that both $y_0$ and $y_1$ links are traversed from the same element. So the maximum fanout is $(l - 1) + 2 = l + 1$, which is also the maximum depth.
Example 4:

Figure 4.1 shows the search tree for the interval \([18, 28]\) = \([(10010)_2, (11100)_2]\), rooted at an element within \([a, b]\) (see Figure 3.4). A maximal \(SI = [20, 27] = [(10100)_2, (11011)_2]\); \(F = \{0, 1, 3\}\) and \(y_0 = y_1 = 2\). Figure 4.2 shows the search tree for the interval \([c, d] = [18, 26] = [(10010)_2, (11010)_2]\) (see Figure 3.6) rooted at an element outside \([a, b] = [20, 23] = [(10100)_2, (10111)_2]\). Here \(F = \{0, 1\}\), \(y_0 = 2\) and \(y_1 = 3\). Note that there are no links to any node outside the intervals searched. Thus such trees can be set up concurrently for non-overlapping intervals without any contention or deadlock. Also, the algorithm ensures that no element within a search interval is reached twice. So the trees can be used for broadcasting messages to all elements in an \(SI\) without having to check whether an element has already received a copy.

4.2 Invariance of PBTs

In the following, we explore additional properties of PBTs in terms of various operations as defined by Ho and Johnsson [23] for spanning binomial trees on a
Figure 4.2: Search tree for interval of [18, 26] with $R = 26$.

hypercube. In particular, we explore the invariance of PBTs under the *translation*\(^1\) operation. By *invariance*, we mean that the resulting tree is also a PBT.

A PBT remains invariant with respect to the translation operation. If we translate the PBT by a node address, then we get a PBT for some interval, but not necessarily the old interval. However, if the new PBT has its root within the old interval, then it spans the old interval. This follows from the property that for any element $\alpha \in SI$, there exists a PBT rooted at $\alpha$ that spans exactly those elements that constitute the set. Also, translation preserves the dimension of the edges. So the task is just to determine the node address by which the PBT is to be translated. This can be given by $G(R) \oplus G(B)$, where $G(R)$ is the Gray code address of the root of the original PBT and $G(B)$ is the Gray code address of the root of the translated PBT.

It is to be noted here that translation does not preserve the search tree for an interval that is not an SI. An example is shown in Figure 4.3 where the PBT in Figure 4.1 is translated by (00010). The translated tree no longer spans a continuous

---

\(^1\)Translation of a graph $G = (V, E)$ by a node address $S$ is defined as $Tr(G) = (Tr(V), Tr(E))$ such that $Tr(V) = \{x \oplus S | x \in V\}$ and $Tr(E) = \{(x \oplus S, y \oplus S) | (x, y) \in E\}$. 
Figure 4.3: Translation may not confine a search tree to an interval.
CHAPTER 5

EXTENSIONS OF PBTs

5.1 Processing of Multidimensional Data

The search tree algorithm can be easily extended to cater to multidimensional data for which there is a linear ordering in each dimension. For example, consider a two dimensional array of $M \times N$ elements indexed graphically as $(x, y)$ pairs. Suppose we want to span the region $([c,d], [e,f])$ from any point $(r, s)$ within that region. We first form a spanning tree in the $X$ direction so that elements $(c, s)$ through $(d, s)$ are reached. Each of these elements form the root of a spanning tree to cover the elements of the region with the same $X$ coordinates. Note that the trees in the $Y$ direction can be grown as soon as a "root" element is reached, so that concurrency is maintained.

If $M$ and $N$ are powers of two, then the spanning trees in the $X$ direction use only the most significant $\log M$ dimensional links of the hypercube, while the trees in the $Y$ direction are formed using the least significant $\log N$ dimensional links. The Gray code mapping is done separately for the $X$ and $Y$ coordinates. From Corollary 1, this does not lead to any conflict of shared links. Algorithm 2 can be used for generating search trees with two-dimensional, and in general, multidimensional data, with the modification that an element initiates a tree in the $Y$ direction as soon as it gets incorporated in some tree growing in the $X$ direction.

Therefore, we have:

Lemma 2 A region of $M \times N$ elements can be spanned by a search tree rooted at any point in this region. This tree has a maximum depth of $(\lceil \log M \rceil + \lceil \log N \rceil + 2)$, and maximum fanout of $\lceil \log M \rceil + \lceil \log N \rceil$. 

23
5.2 Coarser Granularity

In practical situations, the number of elements \( n \) is much larger than the number of processors \( (P) \) available. In such cases, there is a many-to-one relation between an element index and a processor index. Let each processor store two additional variables: \( ind_{\text{min}} \) and \( ind_{\text{max}} \), denoting the minimum and maximum index values of the elements that are mapped onto it. Then an element with index \( I \) is mapped onto processor \( P \) if \( ind_{\text{min}}(P) < I < ind_{\text{max}}(P) \). Let \( ind_{\text{max}}(P) \) be the general index for element \( I \).

For static sets, we can define and maintain a gross linear ordering on the elements such that for any two processors, the indices of all elements mapped onto one of them are less than the index of any element mapped onto the other [30]. In this case, the search trees will be defined on the general indices of the elements. All properties from Chapters 3 and 4 are carried over, except that two non-overlapping intervals may now share a common processor if the maximum element of one interval is mapped onto the same processor as the minimum element of the other. Our scheme can be extended for such coarser granularity in a straightforward way.

For dynamic sets such as dictionary machines [31] with two or more keys that can insert and delete elements, the elements may not get evenly distributed over the processors. A load balancing technique is required to avoid overflows in the number of records or saturation in message traffic at individual nodes. If the difference in load between neighbors in the linear order exceeds a threshold, one can balance this local differential by transferring some elements and adjusting the \( ind_{\text{min}} \) and \( ind_{\text{max}} \) values. More elaborate schemes for load balancing are beyond the scope of this paper [10].
5.3 Faulty Hypercubes

A hypercube is called *faulty* if it contains any faulty processor or link. In order to achieve sustained high performance computing, fault tolerance in hypercubes is very crucial. For hypercubes of large dimensions, the number of processing elements is very large and the probability of occurrence of faults is also very high. A network is *robust* if its performance does not decrease significantly when its topology changes. Since efficient cooperation between the nonfaulty processors is desirable, one measure for robustness is the *connectivity* of a network, which is defined as the number of node or link failures that can be allowed without disrupting the system. Thus, high connectivity prevents the nonfaulty processors from being disconnected. Also, in highly connected networks like the hypercube, the diameter of the *surviving network*, i.e. the subnetwork formed by the nonfaulty processors and links can be reasonably bounded, if the number of faults does not exceed \( n \) [16]. The hypercube network has been proved to be very robust [7, 37] and to disconnect the hypercube into two components requires at least \( n \) faults. Several researchers have developed algorithms for solving a wide range of problems in the presence of processor and/or communication link faults in hypercubes [4, 7, 22, 33].

In this section, MIMD and SIMD algorithms for constructing PBTs in faulty hypercubes are presented. The construction of a PBT involves spanning all nodes in a subcube of a hypercube. Hence, it essentially spans a hypercube of fewer dimensions than the original hypercube. Since we are interested in the concurrent set up and manipulation of several disjoint trees, it is desirable to only use the available (non-faulty) nodes within a subcube of a faulty hypercube and span that subcube. The following assumptions are made about the \( n \)-dimensional hypercube:

- There are at most \( n - 1 \) faulty nodes.
- There are no faulty links.
The set of faulty nodes are known globally. This can be easily obtained through broadcast algorithms.

5.3.1 Free Dimensions

The concept of free dimensions, as suggested by Raghavendra et. al. [36] can be used to broadcast within faulty hypercubes. A dimension is said to be free if no pair of nodes across that dimension link are both faulty. In an n-dimensional hypercube, with number of faulty nodes \( f \leq n \), there exists at least \( n - f + 1 \) free dimensions [36]. Also, the dimensions of an n-dimensional hypercube can be ordered as \((d_1, ..., d_n)\) such that for every \( k \), where \( 1 \leq k \leq n \), every subcube induced by the dimensions \((d_1, ..., d_k)\) contains at most \( k - 1 \) faulty nodes [34]. An outline of the proof is given here for the sake of understanding.

Assume that the set of faulty nodes \( F \) is ordered arbitrarily, as \( F = (f_1, f_2, ..., f_{n-1}) \), where \( f_i \) is the \( i \)th faulty node. Another fact can be proved using induction: there is an ordering \((i_1, i_2, ..., i_{n-2})\) of some \( n - 2 \) dimensions such that for every \( j \), \( 1 \leq j \leq n - 2 \), no two faulty nodes among \( f_1, ..., f_{j+1} \) agree in their values of the \( j \)-bit vector in positions \( i_1, ..., i_j \). The ordering \((i_1, i_2, ..., i_{n-2})\) can be constructed as follows:

Define the \( j \)-dimension as:

1. Choose \( i_1 \) to be any dimension in which \( f_1 \) disagrees with \( f_2 \).
2. For \( j \geq 2 \), if \( f_j \) agrees with some earlier faulty node \( f_l \) \( (l \leq j) \), and let \( i_j = t \).
3. If there is no such \( f_l \), then choose for \( i_j \) any dimension that is not already chosen.

It is easy to prove by induction [34] that the above construction yields the desired ordering of \((i_1, i_2, ..., i_{n-2})\).
To construct the ordering \((d_1, d_2, \ldots, d_n)\) of dimensions, we reverse the ordering of the \(i_j\)'s. Choose \(d_1\) and \(d_2\) to be the two dimensions that were not chosen in the above procedure and for \(j \geq 3\), \(d_j = i_{n-j+1}\). Consider dimensions \(d_1, d_2, \ldots, d_j\). If these bit positions are varied over all possible values, while fixing the remaining bits arbitrarily, a \(j\)-subcube with at most \(j - 1\) faults is obtained. This is because, no two faulty nodes among \(f_1, f_2, \ldots, f_{n-j+1}\) agree on their values of the \((n - j)\)-bit vector in positions \(d_{j+1}, \ldots, d_n\). Thus, every \(j\)-cube formed by varying the dimensions \(d_1, d_2, \ldots, d_j\) contains at most \(j - 1\) faults, i.e. the faulty nodes \(f_{n-j+2}, \ldots, f_{n-1}\) together with one of \(f_1, f_2, \ldots, f_{n-j+1}\).

To make the above analysis consistent with our notations, we invert the ordering of the dimensions to yield \((d_n, \ldots, d_1)\) and renumber them as \((n - 1, \ldots, 0)\). Thus, the fact can be restated as: The dimensions of an \(n\)-dimensional hypercube can be numbered as \((n - 1, \ldots, 0)\) such that for every \(k\), \(0 \leq k \leq n - 1\), every subcube induced by the dimensions \((k - 1, \ldots, 0)\) contains at most \(k - 1\) faulty nodes. Hence, the dimensions of a faulty hypercube can be reordered so that any subcube spanned by a PBT with flip dimensions \(j, \ldots, 0\) contains at most \(j\) faults.

5.3.2 MIMD Algorithms for PBT Construction

Raghavendra et al. [36] have proposed the following algorithm for MIMD broadcasting in faulty hypercubes. Given \(f \leq n - 1\) faults, two free dimensions \(x, y\) can be found such that each 2-subcube spanned by \(x, y\) contains at most one fault. For a root node \(R\), at least one neighbour across \(x\) or \(y\) is fault-free. Without loss of generality, assume that the neighbour across \(x\) is fault-free, and \(R\) sends a message to this neighbour across \(x\). The \(n\)-dimensional hypercube, \(Q_n\), can be broken into two \(Q_{n-1}\)'s along \(x\). There are two cases to be considered.

Case 1: When one of these \(Q_{n-1}\)'s is fault-free, the message can be broadcast inside it and finally sent to the other faulty \(Q_{n-1}\) along \(x\). In this case, \(n + 1\) steps are
required to broadcast.

Case 2: In this case where each $Q_{n-1}$ has some faults, then by induction, broadcasting can be complete in each $Q_{n-1}$ in $n$ steps. Thus, it takes a total of $n + 1$ steps to broadcast from $R$ to all fault-free nodes in the $Q_n$. Since the diameter of the faulty $Q_n$ is $n + 1$ with number of faults $f \leq n - 1$, this scheme is optimal.

Since the PBT construction merely spans a smaller cube in the $n$-dimensional hypercube, the same idea can be used here.

- **Type I PBT**

  A PBT of Type I spans the subcube generated by dimensions $j, ..., 0$. The dimensions of the hypercube can be reordered so that this subcube contains at most $j$ faults. Hence, essentially a $(j + 1)$-dimensional hypercube with at most $j$ faults is being spanned. Thus, the preceding MIMD algorithm can be used to construct a PBT in $j + 2$ steps.

- **Type II PBT**

  A PBT of Type II spans the subcube generated by dimensions $k, j, ..., 0$, where $k > j + 1$. Essentially, this means spanning two subcubes generated by dimensions $j, ..., 0$ and whose corresponding nodes are neighbors in dimension $k$. As explained earlier, two free dimensions $x$ and $y$ can be found in a subcube generated by $j, ..., 0$ such that each 2-subcube generated by $x$ and $y$ contains at most one fault. For the root node, $R$, at least one neighbor across $x$ and $y$ is fault free. We can assume that the neighbour across dimension $x$ is fault free. Let this neighbour be denoted by $S$. There are two cases to be considered.

  Case 1: The neighbour of the root node, $R$, across dimension $k$ may be faulty. Let this node be denoted by $R(k)$.

  Case 2: The neighbour of node $S$ across dimension $k$ may be faulty. Let this node be denoted by $S(k)$. 
However, both cases cannot be true at the same time since $R(k)$ and $S(k)$ are neighbors along dimension $x$, which is a free dimension.

In the first case, root node $R$ can send a message to $R(k)$ in one step. The two respective $(j + 1)$-dimensional subcubes can be spanned in an additional $j + 2$ steps using the idea for a Type I PBT. Thus, a total of $j + 3$ steps is required. In the second case, it will take two steps for the message to go from $R$ to $S$ and then to $S(k)$. The two respective $(j + 1)$-dimensional subcubes can be spanned in an additional $j + 2$ steps. Thus, a total of $j + 4$ steps is required. Hence, in the worst case a Type II PBT can be constructed in $j + 4$ steps.

- **Search Trees**

  Search trees may traverse two more dimensions, $y_0$ and $y_1$, in addition to those traversed by a PBT to reach out to nodes outside the spanning interval (SI). The same idea for the $k$th dimension as in a Type II PBT can be used for $y_0$ and $y_1$. But, there is a difference in this case in that not all nodes in the two dimensions may be traversed. Hence, we are not looking at a complete subcube. And, it may be possible that the neighbours of $R$ and $S$ along $y_0$ and $y_1$ may not be needed to be spanned at all. We can try to reach as many neighbouring nodes along $y_0$ and $y_1$ as possible. But, it may happen that there is one node outside the SI that is reachable from only one faulty node within the SI. In such cases, however, there is no way we can reach out to the node outside the SI unless we relax the assumption of creating disjoint trees for disjoint intervals.

5.3.3 **SIMD Algorithms for PBT Construction**

A simple algorithm for broadcasting from a source node, $S$, in a fault-free hypercube is as follows:

Execute $n$ steps, such that during the $i$-th step, every node sends its message to
dimension $n - i$. This algorithm is described by the sequence $(n - 1, ..., 1, 0)$.

Although this algorithm does not work if there are one or more faulty nodes, it can be modified to work in the case of $f \leq n - 1$ faulty nodes as follows:

Execute $n + f - 1$ steps, such that the sequence of dimensions along which the non-faulty nodes send messages is $n - 1, n - 2, ..., 0, n - 1, n - 2, ..., n - f, n - f - 1$.

The above algorithm correctly broadcasts messages from an arbitrary source node to every other node in the presence of up to $n - 1$ faults [42].

Raghavendra and Sridhar [34] have recently proposed five SIMD algorithms in a stepwise refinement manner for broadcasting in faulty hypercubes. These algorithms require, respectively, $n + 3\log n$, $n + 2\log n + 7$, $n + \log n + O(\log \log n)$, $n + \log n + 5$, and $n + 5$ time steps. The first algorithm using $n + 3\log n$ closely follows the first two SIMD algorithms described in this section. However, the dimension sequence is interrupted with periodic "fix-up" phases. The algorithm can be divided into phases which are termed as either a transmission phase or a flooding phase. The latter phase, in which fix-up occurs, is intended to allow non-faulty nodes that would normally have received the message, but did not do so because of the presence of other faulty nodes, to catch up. The algorithm with minor notational modifications and reordering of dimensions is as follows:

(1) Construct the ordering of the dimensions $(n - 1, ..., 0)$ such that any $k$-dimensional subcube generated by dimensions $(k - 1, ..., 0)$ contains at most $k - 1$ faults.

(2) Broadcast from $S$ to the $k$-subcube containing $S$ and induced by dimensions $(k - 1, k - 2, ..., 0)$ (using the first SIMD algorithm described in this section for broadcasting in faulty hypercubes).

(3) Set $i \leftarrow k$.

repeat

(3a) transmission phase
Let \( m = \min(i + 2^k - k - 1, n) \).

Each node transmits along the dimensions \((m - 1, m - 2, ..., i - 1)\).

(3b) flooding phase

Each node transmits along the dimensions \((k - 1, ..., 0)\) twice.

(3c) if \( m \geq n \), then terminate this loop.

(3d) Set \( i \leftarrow i + 2^k - k \).

forever.

An analysis of this algorithm shows that it is correct and takes \( n + k(\lceil (n - k)/(2^k - k) \rceil + 2) \) steps [34]. In the special case, where \( k = \log n \), the algorithm uses \( n + 3\log n \) steps. Four other algorithms described by Raghavendra and Sridhar [34] are refinements of the above algorithm, each improving in the number of time steps. Since the PBT spans a smaller cube of the hypercube, the preceding algorithm (or its refinements) can also be used for PBT construction.

- **Type I PBT**

  A PBT of Type I spans the subcube generated by dimensions \( j, ..., 0 \). The dimensions of the hypercube can be reordered so that this subcube contains at most \( j \) faults. Hence, essentially a \((j + 1)\)-dimensional hypercube with at most \( j \) faults is being spanned, and the preceding SIMD algorithm can also be used to construct the PBT in \( n + 3\log n \) steps.

- **Type II PBT**

  A PBT of Type II spans the subcube generated by dimensions \( k, j, ..., 0 \), where \( k > j + 1 \). Essentially, this means spanning two subcubes generated by dimensions \( j, ..., 0 \) and whose corresponding nodes are neighbors in dimension \( k \). This case is similar to the one for a Type I PBT, except that all nodes have one extra step of transmitting along dimension \( k \) in the beginning of the algorithm. Hence, the PBT
can be constructed in $n + 3\log_2 n + 1$ steps.

- **Search Tree**

  Here again, there are two extra dimensions $y_0$ and $y_1$ to be considered. As in the case for a Type II PBT, this should require two more steps. But this method suffers from the same problems as the corresponding case for MIMD algorithms for similar reasons.

5.4 **Generalized Hypercubes**

5.4.1 **$k$-ary $n$-cube**

The binary $n$-cube is a special case of the family of $k$-ary $n$-cubes, which are cubes with $n$ dimensions and $k$ nodes in each dimension. The total number of nodes in such a cube is $N = k^n$. Most parallel computers have been built using networks that are either $k$-ary $n$-cubes or their isomorphic structures such as rings, meshes, tori, direct and indirect binary $n$-cubes [5, 13].

A node in the $k$-ary $n$-cube can be represented by an $n$-digit address $d_{n-1} \ldots d_0$ in radix $k$. The $i^{th}$ digit, $d_i$, of the address represents the node's position in the $i^{th}$ dimension, where $0 \leq d_i \leq k - 1$. Two nodes with addresses $d_{n-1}^{(1)} \ldots d_0^{(1)}$ and $d_{n-1}^{(2)} \ldots d_0^{(2)}$ are neighbors in the $i^{th}$ dimension if and only if either $d_i^{(1)} = (d_i^{(2)} + 1) \mod k$ or $d_i^{(2)} = (d_i^{(1)} + 1) \mod k$. Figure 5.1 and Figure 5.2 provide examples of a 6-ary 3-cube and a 8-ary 2-cube.

5.4.2 **Radix-$k$ Gray Code**

Generalized Gray codes and their properties have been studied in [9]. These codes are also applicable to $k$-ary $n$-cubes.

A fundamental ordering $\beta$ of the digits in the set $\mathcal{K} = \{0, 1, ..., k - 1\}$ is any permutation of the digits. The reflected fundamental ordering $\gamma$ of the digits in
Figure 5.1: 6-ary 3-cube

Figure 5.2: 8-ary 2-cube
\( \kappa \) is a permutation given by

\[
\gamma(i) = \beta(k - i - 1). 
\] (5.1)

Without loss of generality, the fundamental ordering \( \beta(i) = i \) (identity permutation) is assumed. All results carry over to any choice of \( \beta \).

Let \( I = [0d_{n-1}d_{n-2}...d_1d_0] \), where \( 0 \leq I \leq k^n - 1 \). Then the reflected Gray code mapping of \( I \) is denoted as \( G^k(I) = g_{n-1}g_{n-2}...g_1g_0 \). As shown below, the encoding and decoding functions for these codes depend on the parity of \( k \). (Note that \( d_n = 0 \) always.)

Case 1: Let \( k \) be even. Then, for \( 0 \leq i \leq n - 1, \)

\[
g_i = \begin{cases} 
\beta(d_i), & \text{if } d_{i+1} \text{ is even} \\
\gamma(d_i), & \text{if } d_{i+1} \text{ is odd}
\end{cases}
\]

and for \( 1 \leq i \leq n, \)

\[
d_{n-i} = \begin{cases} 
\beta^{-1}(g_{n-i}), & \text{if } d_{n-i+1} \text{ is even} \\
\gamma^{-1}(g_{n-i}), & \text{if } d_{n-i+1} \text{ is odd}
\end{cases}
\]

Case 2: Let \( k \) be odd. Then, for \( 0 \leq i \leq n - 1, \)

\[
g_i = \begin{cases} 
\beta(d_i), & \text{if } \sum_{j=i+1}^{n} d_j \text{ is even} \\
\gamma(d_i), & \text{if } \sum_{j=i+1}^{n} d_j \text{ is odd}
\end{cases}
\]

and for \( 1 \leq i \leq n, \)

\[
d_{n-i} = \begin{cases} 
\beta^{-1}(g_{n-i}), & \text{if } \sum_{j=n-i+1}^{n} d_j \text{ is even} \\
\gamma^{-1}(g_{n-i}), & \text{if } \sum_{j=n-i+1}^{n} d_j \text{ is odd}
\end{cases}
\]

Example 5: Consider the linear ordering \( a_0, a_1, ..., a_{15} \) and the Gray code mappings of their indices. The processors onto which they are mapped into a 4-ary 2-cube are shown in Figure 5.3. At each node the vectors not in parentheses are the indices
and those in parentheses are the corresponding Gray codes (addresses of the nodes). The encircled values are the indices in decimal. It can be seen from Figure 5.3 that the Gray code embedding ensures that adjacent elements in the linear ordering are mapped onto adjacent processors.

5.4.3 PBT Extension to k-ary n-cubes

We generalize the concept of a PBT to include trees spanning a subspace of the k-ary n-cube. As before, we fix the values of the digits for those dimensions that are not traversed, and allow all possible combinations of values for the other digits. The construction is similar to the PBT construction except that there may be more than two nodes to be spanned along a particular dimension.

Whenever a node has to traverse a particular dimension, it can reach out to its two neighbors along that dimension. Other nodes along that dimension are spanned successively by these two neighbors, and so on. The set \( K = \{0, 1, \ldots, k-1\} \) is initially stored by all nodes. One additional parameter, called \( K_{set} \), is sent to each node which performs a receive\((dim, flip-set, K_{set})\) from its parent. The \( K_{set} \) contains the values of the digits in dimension “dim” in the addresses of the nodes that
are yet to be spanned. The node checks whether any of these are its neighbors. If so, it does a send to that neighbor with the $Kset$ deleted of the newly spanned value. Then the node proceeds to the original part of the PBT construction. There are two changes in this extension. For each $D \in flip-set$, there are two neighbors that can be spanned and for each one the $Kset$ is assigned to $K - \{g_D^j, (g_D^j + 1) \mod k, (g_D^j + k - 1) \mod k\}$, where $g_D^j$ is the value of the digit at the $j^{th}$ dimension in the node’s address. The formal algorithm for generating the PBT extension to $k$-ary $n$-cube is given below.

Algorithm 3: PKT-GENERATE ($R, F$);

(1) send($\infty, F, K$) to processor $G(R)$ (* initialization *)

(2) for each processor $g_n^i \ldots g_0^i$

    receive ($dim, flip-set, Kset$);

    if (($g_{dim}^i + 1) \mod k \in Kset$ then
        send($dim, flip-set, Kset - \{(g_{dim}^i + 1) \mod k\}$)
        (* send to any other processor along the same dimension *)
    endif

    for all $D \in flip-set$ in parallel do
        send($D, \{e : (e \in flip-set) \land (e < D)\}, K - \{g_D^i, (g_D^i + 1) \mod k, (g_D^i + k - 1) \mod k\}$
        to neighbors in $D^{th}$ dimension.
        (* claim two neighbors along $D^{th}$ dimension as children *)
    enddo.

Example 6: Figure 5.4 shows the tree rooted at (123) spanning interval [16, 31] with the flip set $F = \{0, 1\}$ in a 4-ary 3-cube. The tree spans the physical subspace $1xx$. 
In Figure 5.4, node (123) flips along the $0^{th}$ dimension, reaching out to nodes (120) and (122). To each one of them it sends the $Kset$ as $\{0, 1, 2, 3\} - \{0, 2, 3\} = \{1\}$. That is, 1 is yet to be spanned in the $0^{th}$ dimension. After (122) and (120) receive $(0, \phi, \{1\})$ from the parent (123), node (120) finds $(g_D + 1) \mod k = 1 \in Kset = \{1\}$, and hence spans node (121).

**Theorem 5** The maximum depth of the pruned tree corresponding to a $k$-ary $n$-cube is $(k - 2) \log_k N$.

**Proof:** The maximum depth of the tree will be one more than the maximum depth of the subtree which starts traversing the maximum flip dimension. This subtree will successively traverse all lower dimensions in the flip set. Assume that there are $f$ flip dimensions. When a node traverses a flip dimension, its two neighbors can be reached. In the worst case, starting from one of these neighbors there will be a path spanning the other $(k - 3)$ nodes along that dimension. Since $f$ dimensions are
traversed in the subtree, in the worst case, the longest path can have \((k - 2)f\) nodes. Hence the maximum depth of the entire tree is \((k - 2)f\).

Since there are \(f\) flip dimensions and a digit in each flip dimension can take \(k\) values, the number of nodes in the entire tree is given by \(N = k^f\). Hence the theorem. \(\square\)

**Remark:** This is not the maximum depth of a PBT, since by letting \(k = 2\), the maximum depth is zero. This is due to the fact that the longest path considered here does not exist in a PBT.

Let us now see which intervals can be embedded in PKTs.

**Lemma 3** Spanning intervals (SIs) of Type II do not exist in a \(k\)-ary \(n\)-cube.

**Proof:** Choose a root \(R = d_{n-1}^r \ldots d_{j+1}^r d_j^r \ldots d_0^r\) in a \(k\)-ary \(n\)-cube. Also, let the flip set \(F = \{0, 1, \ldots, k < j - 1, j\}\). An interval given by \([d_{n-1}^r \ldots d_{j+1}^r 0(k - 1) \ldots (k - 1)0 \ldots 0, d_{j+1}^r (k - 1)0 \ldots 0(k - 1)\ldots (k - 1)]\) will not be spanned because it is not possible to reach elements with \(d_j^r \ldots d_{k+1}^r = 10 \ldots 0\) or \(20 \ldots 0\). \(\square\)

**Lemma 4** All intervals \([nk^j, (n + 1)k^j - 1]\) are SIs of Type I and have length \(k^j\).

**Theorem 6** Any interval \([r, s]\) with length \(k^j - 1 \leq L(r, s) < k^{j+1} - 1\) contains an SI of Type I having length \(k^{j-1}\) or more.

**Proof:** For any interval \([r, s]\), we will give the maximal SI in it. Let

\[
\begin{align*}
r &= d_{n-1}^r \ldots d_{j+1}^r d_j^r d_{i-1}^r x \ldots x, \\
s &= d_{n-1}^s \ldots d_{j+1}^s d_j^s d_{i-1}^s x \ldots x,
\end{align*}
\]

where \(d_{n-1}^r \ldots d_{i+1}^r = d_{n-1}^s \ldots d_{i+1}^s = M\), say, and \(d_j^r < d_j^s\). Then one maximal SI is given by \([a, b]\) such that

1. \(a = d_{n-1}^r \ldots d_{j+1}^r d_j^r \ldots d_{i-1}^r x \ldots x,\)
2. \(b = d_{n-1}^s \ldots d_{j+1}^s d_j^s \ldots d_{i+1}^s = M\),
3. \(a < b < s.\)
\[ a = M\hat{d}_r^i d_{i-1}^r 0 \ldots 0 + p, \]

\[ b = M\hat{d}_r^i d_{i-1}^r (k - 1) \ldots (k - 1) + p, \]

and

\[
p = \begin{cases} 
    k^{i-1} & \text{if } r \neq M\hat{d}_r^i d_{i-1}^r 0 \ldots 0 \\
    0 & \text{otherwise.}
\end{cases}
\]

However, this SI is not unique, because \( p \) can take different values given by

\[ p = \delta k^{i-1}, \text{ where } \delta \in \{0, 1, \ldots, k - 1\} \text{ and so chosen that } b \leq s. \]
CHAPTER 6

APPLICATIONS OF PBTs

6.1 Concurrent Processing of Multiple Sets

Recently, a strong case has been made for considering sets and arrays as the basic data structures in parallel programming [11]. The concurrent data structure described in this thesis is useful for applications that are based on an implicit ordering (max, min, successor, geometric position) and/or demand operations of broadcasting, searching or merging on multiple data sets. This is because concurrent manipulation of static sets can be formulated as operations on disjoint segments of a linear chain embedded in the hypercube. To achieve this, we define a linear ordering, <, on the set \( S \) consisting of all the \( N \) elements. Thus, \( \forall a, b \in S \), either \( a < b \) or \( b < a \) unless \( a \) and \( b \) are the same element. For example, in a dictionary, sorting alphabetically according to the key value of each item will lead to such an ordering provided the keys are distinct. Similarly, suppose we have \( q \) static sets, \( S_1, \ldots, S_i, \ldots, S_q \), with arbitrary sizes \( m_1, \ldots, m_i, \ldots, m_q \) respectively. For the \( j \)th element in the \( k \)th set, we can assign a new index \( I \) given by:

\[
I = j + \sum_{h=1}^{k-1} m_h.
\]  

These new indices impose a total order over elements in all sets [28]. Since broadcast trees can be formed without conflicts on disjoint segments of a linear chain, we can have one tree corresponding to the elements of each set, and operate these search/broadcast trees concurrently.
6.1.1 FIND Operations on Sets

One of the most useful operations on sets is FIND, which checks if a given key element belongs to a set $S$ consisting of $N$ elements. An $m$-way find can be performed in which $m$ find operations can be executed in parallel as follows.

An arbitrary node $B$ is fixed as the source node. With $B$ as the root, a PBT is formed using the leading $\log m$ bits as flip dimensions and spanning an interval $m$. Each node so spanned gets one key element to be searched for. With each such node as the root, a PBT is formed using the $\log N$ least significant bits as flip dimensions and spanning an interval of $N$. Each processor in an interval of $N$ stores exactly one element of the set, $S$. In this way, each PBT performs one find operation on the set, $S$. To avoid conflict of nodes the number of bits in node $B$ should be at least $\log m + \log N = \log mN$. That is, the number of processors required is $P = mN$.

The time for $m$ find operations is $T = O(\log m + \log N)$, which is proportional to the depth of the search tree. This time can be compared to $O(m \log N)$, which is the sequential time required for $m$ find operations on a set of $n$ elements.

A wide class of graph algorithms (connected components, minimum spanning tree etc.) and algorithms in computational geometry (convex hull, line of sight) use a scan operation as a primitive [8]. Segmented scan algorithms can be implemented as broadcast/merge operations on multiple sets.

The search trees can be used for concurrent I/O subsystems wherein some of the processing nodes have direct connections to I/O nodes. Suppose the selection of the I/O node sites is made by first imposing a linear ordering on the processors, and then having each I/O node serve a particular segment. Broadcast and multicast are common I/O operations, and can be achieved efficiently using the search trees. Also, the effect of uneven traffic can be alleviated by dynamically changing the segment size allocated to the various I/O nodes and quickly establishing the search trees for the new configuration.
6.2 Ordered h-level graphs and Computational Geometry

Dehne and Rau-Chaplin [14] proposed ordered h-level graphs \(^1\) as suitable data structures for implementing efficient parallel algorithms for problems in computational geometry on hypercube multicomputers. In particular, it is shown that for such a graph with \(N\) nodes, \(O(N)\) search processes can be efficiently executed, and in an \(m\)-way search, an arbitrary number of search queries can simultaneously access the same hypercube node. Additionally, the authors have applied the \(m\)-way search to implement a segment tree and solve the next element search, trapezoid decomposition and triangulation problems in \(O(\log^2 N)\) time using \(O(N \log N)\) processors.

6.2.1 \(m\)-Way Search on Ordered h-level Graphs

Given a directed graph \(G = (V, E)\) with a vertex-set \(V\) and an edge-set \(E\), an ordered h-partitioning of \(G\) is a partitioning of \(V\) into an ordered sequence of \(h\) disjoint sets \(L_1, \ldots, L_h\) together with an ordering of the elements in each subset \(L_i\), for \(1 \leq i \leq h\). For every vertex \(v \in L_i\), we define \(\text{Level}(v) = i\), \(\text{LevelIndex}(v)\) as the rank of \(v\) with respect to the ordering of the vertices in \(L_i\), and \(\text{Index}(v) = (\sum_{j=1}^{i-1} |L_j|) + \text{LevelIndex}(v)\).

Now a directed, acyclic, and layered graph \(G\) is called an ordered h-level graph if

1. Every node of \(G\) has an out-degree of at most \(k = O(1)\),

2. There exists an ordered h-partitioning \(L_1, \ldots, L_h\) of \(G\) such that

   (a) every source of \(G\) is contained in \(L_1\),

   (b) if \((v, w)\) is an edge, then \(\text{Level}(w) = \text{Level}(v) + 1\), and

---

\(^{1}\)Such graphs are acyclic, planar, and include \(k\)-ary search trees for \(k = O(1)\).
(c) Two edges \((v, w)\) and \((v', w')\) satisfying \(\text{Index}(v) < \text{Index}(v')\) imply \(\text{Index}(w) \leq \text{Index}(w')\).

In the following we outline an \(m\)-way search algorithm due to Dehne and Rau-Chaplin [14], on an ordered \(h\)-level graph \(G\) with \(N\) nodes. Given a set \(Q = \{q_1, \ldots, q_m\} \subseteq U\) of \(m = O(N)\) queries, the \(m\)-way search problem consists of executing, in parallel, all \(m\) search processes induced by the \(m\) queries. Before proceeding further, let us define a search path for a query \(q \in U\). It is a sequence \((v_1, \ldots, v_h)\) of \(h\) vertices of \(G\) defined by a successor function \(f: (V \cup \{\text{start}\}) \times U \rightarrow \mathcal{N}\) as follows: (i) \(f(\text{start}, q) = \text{Index}(v_1)\), and (ii) \(f(v_i, q) = \text{Index}(v_{i+1})\), for \(1 \leq i \leq h\), where \(\mathcal{N}\) is the set of positive integers.

The input graph \(G\) is stored in a hypercube such that each vertex \(v\) with index \(i = \text{Index}(v)\) is stored in register \(v(i)\) of processor \(PE(i)\). This register contains fields to store a constant amount of data associated with \(v\), its level, levelindex and index. The edges of \(G\) are stored as adjacency lists. The set \(Q\) of \(m\) queries is stored such that every processor \(PE(i)\) stores one query in its register \(q(i)\). The search algorithm is sketched below, and the reader is referred to [14] for details.

**procedure** \(m\)-Way-Search;

(1) \(\text{Phase}_1\) \{match every query with the first node in its search path.\}

(2) \textbf{for} \(x := 2\) to \(h\) \textbf{do}

\(\text{Phase}_x\) \{Match every query with the \(x\)th node in the search path.\}

The \(m\) search processes for all queries are executed in \(h\) phases such that each phase moves all queries one step ahead in its search path. The algorithm permutes the queries and copies some nodes into registers \(v'(i)\) such that at the end of \(\text{Phase}_x\), for \(1 \leq x \leq h\), all queries are stored with respect to the index of the \(x\)th node in their search path; and each processor \(PE(i)\) containing a query \(q\) in its register \(q(i)\) contains in its register \(v'(i)\) a copy of the \(x\)th node in the search path of
The time complexity of this algorithm is \( T_1 = O(min\{s \log M, \log^2 M\} + h \log M) \), where a hypercube multiprocessor of size \( M = \max\{N, m\} \) is used, and the digraph has \( s \) sources (i.e., nodes at level 1).

6.2.2 Application of PBT to the \( m \)-Way Search

It can be observed that the levels of an ordered \( h \)-level graph span an interval \([1, h]\). Hence each search path for a query \( q = (v_1, \ldots, v_h) \) has to span an interval of \( h \), visiting exactly one node at each level. A PBT spanning an interval of \( h \) has to be formed corresponding to each of the \( m \) search queries. Without loss of generality, we assume that \( m \) and \( h \) are powers of two.

With an arbitrary node, \( B \), as the root, we form a PBT spanning an interval of \( m \) using the most significant \( \log m \) bits as the flip dimensions. Then with each of the nodes spanned in this PBT as the root, we form \( m \) PBTs, each spanning an interval of \( h \) using the \( \log h \) least significant bits as the flip dimensions. Each PBT so formed corresponds to one of the \( m \) search queries.

The roots of all such PBTs are distinct (span an interval of \( m \)). Also the PBTs will not lead to any conflict of shared nodes if the bit address of each node has at least \( \log m + \log h \) bits. Each processor within an interval \( h \) (spanned by the least significant \( \log h \) bits) stores information about all the nodes in a particular level of the ordered \( h \)-level graph. If the \( h \)-level graph has approximately the same number of nodes at every level, this amount of information is constant. If the number of nodes increases with levels, as for a \( k \)-ary tree, a node can form a PBT using its most significant bits for flipping. This PBT can span all the nodes within a particular level. In this thesis, we assume that the number of nodes at every level is approximately the same. If this is not true, the algorithm can easily be modified by using the technique mentioned above. Thus, all the processors within an interval \( h \) together
store the complete information about all the nodes in such a graph. This information is repeated in every other set of processors spanning an interval of $h$ using only the least significant $\log h$ bits as the flip dimensions.

Each processor on receiving a query performs a search in an interval of $h$. The successor rank function will be used by each processor in the PBT spanning an interval of $h$ to search in the nodes stored in it (i.e., all nodes in a level). The proposed algorithm is formally described below.

Algorithm 4: PBT-\textit{m}-Way Search ($R$, $\mathcal{F}_1$, $\mathcal{F}_2$, $Q$);

(1) \textbf{send}($\infty$, $\mathcal{F}_1$, $\mathcal{F}_2$, $Q$) to root processor $G(R)$ (* initialization *)

(2) for all processors do

\hspace{1em} \textbf{receive} (dimension, flip-set$_1$, flip-set$_2$, $Qset$);

\hspace{1em} for all $D \in$ flip-set$_1$ in parallel do

\hspace{2em} \textbf{send}($D$, $\{e : (e \in$ flip-set$_1) \land (e < D)\}$, flip-set$_2$, $\{q_i : (q_i \in Qset) \land (2^j \leq i \leq 2^{j+1} - 1)\}$);

\hspace{3em} (* Here $D$ is the $j^{th}$ dimension in flip-set$_1$ *)

\hspace{2em} enddo;

\hspace{1em} for all $D \in$ flip-set$_2$ in parallel do

\hspace{2em} \textbf{send}($D$, $\emptyset$, $\{e : (e \in$ flip-set$_1) \land (e < D)\}$, $\phi$); (* search *)

\hspace{2em} enddo;

enddo.

In the preceding algorithm, $\mathcal{F}_1$ is the flip set for the first PBT (spanning an interval of $m$), and $\mathcal{F}_2$ is the flip set for the second PBT (spanning an interval of $h$). The first (second) send is for the first (second) PBT, and $Q$ is the set of queries. Each node of the PBT spanning the interval of $m$ performs one query in the $Qset$ received from its parent. The other queries are sent to its children while constructing the PBT spanning the interval of $m$. 
Example 7: Let $h = 4$, $|V| = n = 10$ as in Figure 6.1. Assume that there are $m = 8$ searches, and the number of processors is 32. The number of bits in the processor address is 5. The three most significant bits are used to form an interval of $m = 8$ and the least significant two bits form an interval of $h = 4$. We arbitrarily choose 00000 as the source processor, $B$, and the PBT is shown in Figure 6.2. The thick edges of PBT span an interval of $m$, while the other edges correspond to the PBT spanning an interval of $h$.

6.2.3 Analysis and Comparison

In the following we compare the performance our $m$-way search algorithm with that due to Dehne and Rau-Chaplin [14]. It will be seen that the time required by our algorithm is always less and the processor-time product (i.e., the cost of parallel processing) is no worse.

Total number of hypercube processors in our search algorithm is $p = 2^{\log m + \log h} = mh$, while the total time required to perform $m$ search queries is $T_2 = O(\log h + \log m)$. Let us compare the performance of our algorithm with that due to Dehne and Rau-Chaplin [14]. Two cases are to be considered depending on whether the number of
Figure 6.2: A PBT for illustrating 8-way search.
nodes \((N)\) in the \(h\)-level graph is larger or smaller than \(m\).

**Case 1:** \(N > m\), i.e. \(M = \max\{N, m\} = N\).

In [14], the number of processors used is \(P_1 = N\), whereas the best-case time corresponding to \(s = 1\) is \(T_1 = O(h \log N)\).

In our algorithm, \(P_2 = mh\), and \(T_2 = O(\log h + \log m)\). The relation \(N > m\) implies \(\log m < \log N\). Thus \(T_2 < T_1\).

Furthermore, the processor-time product of the algorithm in [14] is \(P_1T_1 = O(Nh \log N)\), and that in our case is \(P_2T_2 = O(mh \log h + mh \log m)\). Since \(m < N\) and \(h \leq N\), we get \(mh \log m < Nh \log N\) and \(mh \log h < Nh \log N\). Therefore, \(P_2T_1 < P_1T_1\).

**Case 2:** \(m > N\), i.e. \(M = m\).

In [14], \(P_1 = m\), and the best-case time \(T_1 = O(h \log m)\) for \(s = 1\). On the other hand, \(P_2 = mh\), and \(T_2 = O(\log m)\), since \(h \leq N\) and \(N < m\). So \(T_2 < T_1\).

In terms of processor-time product, \(P_1T_1 = P_2T_2 = O(mh \log m)\).

### 6.3 Matrix Computation

As an example, we illustrate the use of binomial trees in the Gaussian Elimination computation used to solve a system of simultaneous algebraic equations \(Ax = b\). For the purpose of the current discussion, we will assume that \(A\) is a square matrix of size \(2^m \times 2^m\) and the cube is of dimension \(2m\) so that each element of matrix \(A\) is stored at a distinct node of the cube.

The mapping of elements of \(A\) onto cube nodes is illustrated in Figure 6.3, where \(m = 2\). The elements of \(A\) are stored in *column-major order*. However, alternate columns are stored in opposite order. The advantage of this storage scheme is that all the elements of a column are mapped to an interval of size \(2^m\) in the Gray
Figure 6.3: Mapping of a 4x4 A matrix onto the hypercube nodes.

code ordering of the nodes of the cube, and at the same time the elements of a row also form an interval of size $2^m$ — the former interval uses the least significant $m$ bits of a node index while the latter uses the most significant $m$ bits. Since both columns and rows define intervals in the Gray code ordering of nodes, by Theorem 2, we can conclude that spanning binomial trees can be mapped over rows and columns of $A$.

An important step in the Gaussian Elimination process is to find a pivot for each column of $A$. Assuming that the matrix is non-singular, there is a pivot in each of the $2^m$ columns. At each step, a pivot is established for one column starting from the left, and the rows involved are $i$ through $2^m$, where $i$ is the column index for which the pivot is being established. From the above it is clear that these column elements are mapped on intervals of processor indices and PBTs can be mapped over them, which makes the pivot selection efficient. Selecting a pivot for column $i$ is initiated at the processor storing the value $a_{ii}$ via a PBT spanning the column elements. The process of establishing a pivot may entail exchanging rows. The processor that initiated the search for the pivot informs those processors storing other elements belonging to its row about the candidate row via a PBT defined over them, which spans from $a_{ii}$ to $a_{i,2^m}$.

Ignoring the cost involved in exchanging rows to establish pivots, the total communication cost for this phase of the algorithm is $O(\sum_{j=1}^{m-1} \log j) = O(m \log m)$. At the end of this first stage of computation matrix $A$ has been transformed into an upper triangular matrix. The next stage involves back-substitution.
The processors storing the diagonal elements compute the solution and propagate it up their column. This is again achieved using the PBTs spanned over the column elements. On substituting these values, the non-diagonal elements transfer the results of their back-substitution to their left neighbors. Thus the communication cost during this stage \( O(\sum_{j=2}^{m-1}(\log j + 1)) = O(m \log m) \).

Although the preceding illustration assumed a square matrix \( A \), the approach can be easily extended to more general situation of rectangular matrices where the number of columns (i.e., number of variables) is greater than the number of rows (i.e., number of equations).

6.4 Image Processing Applications

For two dimensional images, the linear ordering in each dimension can simply be based on the coordinates of each data element (pixel). If a preliminary segmentation of the image yields non-overlapping regions of interest, then search trees can be generated concurrently in each region such that operations such as histogramming, median filtering and image transformation [32] can be done in time logarithmic in the size of the largest region, as evident from Theorem 5. This leads to significant savings if only small fragments of the image are of interest for further processing.

In the above paragraph, it is assumed that the regions of interest are known beforehand. Quadtrees are a popular means of efficient representation of binary images [39]. We now outline how the search trees can be used to generate quadtrees for grey level images. Here, a region is considered homogeneous for segmentation purposes if the intensity value of each pixel in that region is within a (possibly adaptive) threshold of the average value over that region. The original image is divided into four quadrants. For each quadrant, a search tree is used to gather the pixel values within that quadrant, and then broadcast the computed average. If the difference between the local value and the broadcast average is more than the
allowable threshold, then a signal is conveyed to the root. In this case, this quadrant is in turn divided into four sub-quadrants and the process repeated for these new quadrants using four independent search trees. Otherwise, the root broadcasts the unique label representing the quadtree node for that region, to all elements of the region.

The gather and broadcast capabilities of the concurrent search trees have potential applications for several other problems. For example, in [3], a technique for segmentation using multisensor data is given. The authors use a region growing approach using a pyramid data structure [21]. The difference operator within a region in the thermal image is broadcast to the visual pyramid to determine the threshold for segmentation based on the visual image. To use the data structure presented in this paper, an apriori estimate of the number of segments $k$, is made, and $O(k)$ search trees are used to determine regions based on the thermal image, broadcast the difference operator, and obtain the final segmentation based on the visual clues. The effectiveness of this approach can only be determined by actual implementation that indicates whether concurrency obtained through the search trees is eclipsed by the overheads incurred in generating these trees. Also, the techniques introduced here need to be generalized for arbitrarily shaped regions, and to cases where the size of the various regions are to be determined at runtime.
CHAPTER 7

CONCLUSIONS

The search trees described in this thesis provide a flexible and versatile method of performing concurrent broadcast/search operations of distributed data sets. The trees are of optimal depth and are embedded in hypercubes with both expansion and dilation of one. More significantly, they are vertex (and hence edge) disjoint if the logical segments on which they operate are non-overlapping. This is important if circuit-switching or wormhole-routing is used, as such mechanisms are very sensitive to network contention. Since the pruned binomial trees are just a variant of spanning binomial trees, all the methods of communication in spanning binomial trees are applicable to them too.

The extensibility of the pruned binomial trees is high as they can be set up in generalized hypercubes with minor modifications to accommodate the generalized gray codes. This is significant in view of the fact that most parallel computers are built using generalized hypercubes or their isomorphic structures. Pruned binomial trees can also be set up in faulty hypercubes, with number of faults \( f \leq n - 1 \), using either SIMD or MIMD algorithms. They can also be constructed to process multidimensional data.

Pruned binomial trees can be used for concurrent operations of multiple sets, \( m \)-way search with applications to computational geometry problems, matrix computation and image processing. While the applicability of the concurrent data structures described here is wide, their efficient utilization is contingent on choosing the appropriate granularity as determined by the actual multicomputer specifications. Applications of the search trees to dictionary operations needs to be investigated further.


