Algorithms for Incomplete Hypercubes

Venkata K. Prabhala

Western Michigan University

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation

https://scholarworks.wmich.edu/masters_theses/851
ALGORITHMS FOR INCOMPLETE HYPERCUBES

by

Venkata K. Prabhala

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
April 1992
ALGORITHMS FOR INCOMPLETE HYPERCUBES

Venkata K. Prabhala, M.S.
Western Michigan University, 1992

Networked multiprocessing architectures for parallel computation offer an alternative to high cost supercomputing. Recently hypercube has emerged as the most versatile architecture for parallel computations. However, the number of nodes $m$ in a hypercube is a power of 2, $2^d$, where $d$ is the dimension of the hypercube. In practice, it may not be possible to have a complete hypercube because the cost of upgradation is proportional to the number of nodes. Incomplete and Composite hypercubes help remove the exponential node (and hence cost) constraint.

In this thesis we establish the equality of $m$-node composite and incomplete hypercubes. Then we identify a class of algorithms designated Fully Normal Algorithms for implementing a variety of algorithms on composite hypercubes. We define critical size of a composite hypercube to help identify the performance bounds and compute the speedup achieved. Finally, we develop a set of graph algorithms that can be used as building blocks.
ACKNOWLEDGMENTS

It is always a pleasure to thank individuals who contribute directly and indirectly to make an endeavor of this magnitude possible. While it is impossible to thank all of those individually, I feel it is my moral duty to thank at least those persons who have made an immediate and tangible contribution to this effort.

The very first person I wish to thank is my advisor, Dr. Naveed A. Sherwani, whose endless patience and brilliant guidance brought this effort to fruition. I also wish to thank Dr. Alfred Boals and Dr. Ajay Gupta for their advice and comments that proved invaluable in the shaping of this thesis. Next comes the Night Group, the set of excellent young researchers who have provided constant inspiration by their hard work and dedication through the long nights. I am proud to be associated with them.

I warmly acknowledge the emotional support I received from my wife and 5-year-old son. Their patience in the face of my frequent and prolonged absences from home made this effort all the more valuable. My father and mother who made superhuman sacrifices for educating me, my brother and my sisters are constantly in my thoughts for what all they did for me and what all I couldn’t do for them.

Finally I would like to convey my appreciation for the facilities provided by the Department of Computer Science, WMU, under the stewardship of Dr. Donald Nelson.

Venkata K. Prabhala
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Algorithms for incomplete hypercubes

Prabhala, Venkata Krishnarao, M.S.
Western Michigan University, 1992
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ........................................... ii

LIST OF TABLES ................................................. v

LIST OF FIGURES ............................................... vi

CHAPTER

I. INTRODUCTION ............................................... 1

II. PRELIMINARIES ............................................... 7

   2.1 Terminology ............................................... 7

   2.2 Algorithms Related to Hypercubes ...................... 9

      2.2.1 Algorithms for Message Passing in MIMD Hypercubes .. 10

      2.2.2 Fully Normal Algorithms (FNA) ..................... 14

   2.3 Examples of FNA ....................................... 14

III. INCOMPLETE AND COMPOSITE HYPERCUBES ............... 17

   3.1 Renumbering Composite Hypercubes .................... 17

IV. FULLY NORMAL ALGORITHMS ............................... 24

   4.1 Fully Normal Algorithms on Hypercubes .............. 24

      4.1.1 Static and Dynamic FNA for Complete Hypercubes ... 25

   4.2 FNA for Incomplete Hypercubes ....................... 28

   4.3 Analysis of Algorithm FNA.INCOMPLETE ............... 33

      4.3.1 The Allocation Scheme ......................... 34

      4.3.2 Allocation Scheme for $K \leq m$ ............... 34

      4.3.3 Allocation Scheme for $K > m$ ............... 35

V. SIZE/PERFORMANCE TRADEOFFS FOR FNA ................ 39
### Table of Contents – Continued

5.1 Speedup of FNA .................................................. 39
5.2 Analysis of CCH .................................................. 43

VI. GRAPH ALGORITHMS ON INCOMPLETE HYPERCUBES .... 48

6.1 Spanning Forest on IH(m) With Unlimited Memory .......... 48
   6.1.1 Analysis of Spanning_Forest_Incomplete1 ............. 50
6.2 Spanning Forest on IH(m) With Limited Memory .......... 51
   6.2.1 Analysis of Spanning_Forest_Incomplete2 ............. 53
6.3 Other Graph Algorithms for Incomplete Hypercubes ......... 54
   6.3.1 Graph Bipartiteness ......................................... 54
   6.3.2 Fundamental Cycle Set ....................................... 56
   6.3.3 Connected Components and Bridges ....................... 58

VII. CONCLUSIONS .................................................. 60

7.1 Conclusions and Future Directions ............................. 60

BIBLIOGRAPHY ...................................................... 61
LIST OF TABLES

1. Criticality at CH(40) ............................................ 43
2. Variation of CCH With Dimension ....................... 44
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>14-node Composite Hypercube.</td>
<td>4</td>
</tr>
<tr>
<td>2.</td>
<td>16-node Complete Hypercube.</td>
<td>8</td>
</tr>
<tr>
<td>3.</td>
<td>Dimension Collapsing and Computation Cycle.</td>
<td>9</td>
</tr>
<tr>
<td>4.</td>
<td>Routing in Complete Hypercube.</td>
<td>12</td>
</tr>
<tr>
<td>5.</td>
<td>Routing in Incomplete Hypercube.</td>
<td>12</td>
</tr>
<tr>
<td>6.</td>
<td>Static FNA.</td>
<td>15</td>
</tr>
<tr>
<td>7.</td>
<td>Dynamic FNA.</td>
<td>16</td>
</tr>
<tr>
<td>8.</td>
<td>Renumbering CH(5) to IH(5).</td>
<td>20</td>
</tr>
<tr>
<td>9.</td>
<td>Node Allocation Scheme in Complete Hypercube.</td>
<td>27</td>
</tr>
<tr>
<td>10.</td>
<td>Node Allocation in CH(m).</td>
<td>31</td>
</tr>
<tr>
<td>11.</td>
<td>Detailed Example of Node Allocation in CH(m).</td>
<td>37</td>
</tr>
<tr>
<td>12.</td>
<td>Speedup of FNA on CH(32 through 128).</td>
<td>43</td>
</tr>
<tr>
<td>13.</td>
<td>CCH Versus Dimension.</td>
<td>45</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

Parallel processing seeks to improve the speed with which a computation can be done by breaking it into subparts, or subproblems, and concurrently executing as many of these as possible. A parallel computer architecture consists of multiple identical computing nodes and inter-node communication networks. These networks can range from a single shared bus to complex multistage interconnection networks. Several interconnection topologies like Mesh, Hypercube, Perfect Shuffle, Cube Connected Cycles, Butterfly [12] have been proposed and built. A significant body of literature exists on the algorithmic properties of these topologies [4, 12].

One of the major factors in determining system performance is the manner in which the processors communicate with the memory subsystem. Two major approaches are found in contemporary parallel computers. The shared-memory approach employs a single central memory unit to which all processors have direct and rapid access. Contention for this shared memory, however, can result in serious performance loss. An alternative approach is to provide each processor with a local memory to which other processors have slow and indirect access. Such a distributed-memory scheme simplifies the interconnection of massive numbers of processors, but raises new problems in communication efficiency.

Distributed-memory multiprocessors such as hypercube eliminate most of the access contention problems associated with a large shared memory. In recent years, hypercube architecture has emerged as the most versatile architecture for parallel computations [5, 6, 9, 11]. Currently several parallel machines with hypercube architecture like Cosmic Cube, NCUBE, Connection Machine, iPSc/1
and iPSc/2 are commercially available [7, 8, 12, 15]. The popularity of hypercube architecture can be attributed, in a large measure, to its graph properties. Hypercubes are regular and symmetric graphs and have low diameter and high connectivity. Also, several parallel architectures can be efficiently embedded into hypercubes. These graph properties coupled with the availability of commercial machines have led to the development of algorithms in important fields like VLSI, graph theory and image processing [4, 5, 6, 9, 11, 14]. Another feature of hypercubes is the existence of simple and efficient algorithms for internode communication and broadcasting [16]. These algorithms are considered basic building blocks for development of other parallel algorithms.

Hypercube, sometimes referred to as binary n-cube, can be formally defined as follows:

**Definition 1** For a given dimension $d$, a hypercube is one that contains $m = 2^d$ nodes with addresses in the range $0$ through $2^d - 1$ linking every pair of nodes that differ in exactly one bit of their node addresses.

We refer to any hypercube that follows this definition as Complete Hypercube. The physical significance of the dimension $d$ in a complete hypercube of $m$ nodes is that each of the $m$ nodes has one-to-one direct communication links to $d$ neighbors. A direct communication link allows the most efficient communication between the two connected nodes.

However, notice that we can only increment $d$ while building a hypercube architecture. That means the number of nodes $m$ in the network is a power of 2 resulting in large gaps in the sizes of the systems that can be built [10]. In the development of parallel algorithms it is usually assumed that the hypercube is complete. Even when a small number of nodes is required, such algorithms dictate that we have a complete hypercube. For example, if a problem requires 9 nodes, a complete 16-node hypercube must be used. In practice, however, it may not be possible to have a complete hypercube due to operational and commercial
reasons. Therefore, it is important to investigate hypercube architectures which are defined for any value of \( m \).

Katseff proposed in [10] an \( m \)-node Incomplete Hypercube by taking nodes 0 through \( m - 1 \) of a complete hypercube.

**Definition 2** An Incomplete Hypercube \( IH(m) \) with \( m \) nodes is constructed by numbering the nodes from 0 to \( m - 1 \) and linking each pair of nodes whose binary representations differ by exactly one bit.

It is clear that if \( m \) is a power of two, then this scheme describes a complete hypercube. Otherwise, it describes an incomplete hypercube \( IH(m) \). In [10] it was shown that broadcasting and node-to-node communication algorithms for incomplete hypercubes are similar to the algorithms for a complete hypercube. Tzeng in [18] and several other authors investigated a restricted version of Katseff's definition by considering only those \( m \)-node incomplete hypercubes, where \( m = 2^l + 2^p, l > p \). They investigated the capability of this architecture to simulate binary trees and two dimensional meshes. In [1, 2], a generalization of incomplete hypercubes, called Composite Hypercubes is defined.

**Definition 3** Let \( m = 2^d + m' \), where \( 2^{d+1} > m > 2^d > 1 \). An \( m \)-node composite hypercube \( CH(m) \) is defined to be a subgraph of a complete hypercube \( Q_{d+1} \) consisting of two disjoint composite hypercubes \( CH(2^d) \) and \( CH(m') \), where \( CH(2^d) \) is a complete hypercube \( Q_d \). In addition to the edges of \( CH(2^d) \) and \( CH(m') \), the composite hypercube \( CH(m) \) contains the edges of \( Q_{d+1} \) which connect nodes of \( CH(2^d) \) with the nodes of \( CH(m') \).

Figure 1 shows an example of a 14-node composite hypercube. Observe that if \( m = 2^{d_1} + 2^{d_2} + \ldots + 2^{d_k} \), then \( CH(m) \) consists of \( r \) \( d_i \)-dimensional complete hypercubes where every node in \( Q_{d_i} \) is adjacent to exactly one node in each \( Q_{d_j}, i > j \). In [1, 2] it was shown that any two \( m \)-node composite hypercubes are isomorphic and that the composite hypercubes contain maximum number of
edges for \( m \) node subgraphs of complete hypercubes. Composite hypercubes are not restricted to the first \( m \) nodes of a complete hypercube. The basic idea is that a composite hypercube is expressible as a collection of successively smaller complete hypercubes. This collection of complete hypercubes is also required to satisfy an adjacency requirement which maximizes the number of edges in the architecture. Composite and incomplete hypercubes are interesting because they allow upgradability to any number of nodes (in contrast, complete hypercubes need to double in size). It must be noted that definitions 2 and 3 above essentially define subgraphs of complete hypercubes. In [1, 2] it was also shown that composite hypercubes retain most of the interesting graph properties of complete hypercubes.

Since composite and incomplete hypercubes are subgraphs of complete hypercubes, it may initially appear that the many algorithms for complete hypercubes should carry over to these two architectures with minimal modifications. But, as pointed out earlier, composite and incomplete hypercubes lose two important properties of complete hypercubes used in the implementation of most of the graph algorithms: regularity and symmetry. As many known algorithms for hypercubes depend on the regularity and symmetry of complete hypercubes,
in general these algorithms can not be used for composite and incomplete hypercubes. This probably accounts for the absence of algorithmic results on these architectures.

Before we deal with the problem of algorithm design, we must decide which of the two architectures, $IH(m)$ or $CH(m)$, is relatively more powerful as an implementation platform. Then we must look for algorithms that can be used as building blocks for building more complex algorithms and must be able to compute the speedup achieved to compare the quality of the solutions achieved with those for the complete hypercubes.

In this study, we show that $CH(m)$ and $IH(m)$ are architecturally equivalent effectively merging two directions of research. Then we define and implement a class of building block algorithms called *Fully Normal Algorithms* and show that beyond a critical size of the composite hypercube the incomplete architecture has a speedup very close to the ideal linear complete hypercube.

In Chapter II, we introduce terminology relating to the algorithmic manipulations on complete, composite and incomplete hypercube architectures. Then we focus on the message passing primitives needed for hypercubes as the basis for our discussion. We include Katseff's [10] work to show that simple and deadlock-free algorithms exist for message routing and broadcasting in incomplete hypercubes. We explain the concept of Fully Normal Algorithms (FNA) as a major class of algorithms on hypercubes in general.

In Chapter III, we show the architectural equivalence of composite and incomplete hypercubes. We explain the renumbering scheme for composite hypercubes and show that the formal algorithm produces an equivalent incomplete hypercube. This important result enables us to work with composite hypercubes and apply the results directly to incomplete hypercubes and vice versa.
In Chapter IV, we return to a full discussion on Fully Normal Algorithms showing the classification of both the static and dynamic FNA on complete hypercubes. Then we state and prove the case of static and dynamic FNA on incomplete hypercubes impaired by architectural assymmetry.

In Chapter V, we show the necessity for altering the standard definition of speedup in parallel algorithms. Then we compute the size/performance tradeoffs for incomplete hypercubes for FNA and show the existence of performance thresholds below which computation with incomplete hypercubes is feasible but uneconomical. We develop an analytical argument for the relation between the fraction of nodes needed for crossing the performance threshold, and the dimension of the incomplete hypercube.

In Chapter VI we show the computational feasibility of several graph algorithms on incomplete hypercubes that are FNA using a limited-memory-node model with pipelined task allocation. Since the economy involved in reusable software is already well known, we have chosen algorithms that are used as building blocks in several other complex algorithms.

We finally conclude by pointing out several directions of research that need to be pursued for effective use of incomplete hypercube architectures.
CHAPTER II
PRELIMINARIES

In this chapter we review some terminology relating to the structural and algorithmic properties of hypercubes. Since computing nodes in a hypercube network communicate by message passing, we review Katseff’s [10] work on the communication properties of the Incomplete Hypercube network as the foundation for our investigation into the algorithmic properties of incomplete hypercubes.

2.1 Terminology

In terms of topology, a hypercube can be described as follows:

A hypercube of dimension $d$ has $m = 2^d$ processing nodes. The nodes are labeled $0, 1, \ldots, m - 1$ and two nodes are directly linked if and only if their binary addresses differ in exactly one bit position.

See Figure 2 for a pictorial representation of a hypercube of dimension 4. It has 16 nodes and each node has 4 neighbors that have direct communication links. If two node addresses differ in $\alpha \leq d$ bits, then they have $\alpha$ links in their communication path. Since the node addresses have $d = \log m$ bits, the longest communication path has $d$ links. In Figure 2, the longest communication path is 4 links long.

We count the dimensions of a hypercube by a correspondence to the address bits of the nodes. In a hypercube of dimension $d$, each node has $d$ adjacent nodes. The binary address of each node also has $d$ bits. Hence, we can generate the addresses of each of the $d$ adjacent nodes of a given node by complementing each address bit in turn. This means that each node has one address bit per dimension. This brings us to the concept of Dimensional Neighbors. Two nodes are Dimensional Neighbors in dimension $x$ if their binary addresses differ in the
The thick arrows in Figure 2 represent communication in the dimension indicated by the number next to them. Consequently, each node in a hypercube of dimension $d$ has $d$ dimensional neighbors.

Let us define clearly some terms used in the discussion. In a hypercube $Q_d$, we say that dimension $t$, $0 \leq t \leq d - 1$ is collapsed if each node $P_i$ with address $a_d a_{d-1} \ldots a_{t+1} a_{t} \ldots a_0$ sends its results to node $P_j$ with address $a_d a_{d-1} a_{d-2} \ldots a_t a_{t-1} \ldots a_0$. $P_i$ is referred to as the dimensional neighbor of $P_j$. For a hypercube $Q_d$, a sequence of $d$ steps in which successive dimensions $0, 1, \ldots, d - 1$ are collapsed is called a computation cycle. For example, in $Q_3$, when dimension 0 is collapsed, nodes 1, 3, 5, 7 send their data to nodes 0, 2, 4, 6, respectively. Next we collapse dimension 1, and nodes 2 and 6 send their data to nodes 0 and 4, respectively. Finally, node 4 sends its data to 0, thus completing the computation cycle. An example of this data flow is shown in Figure 3. However, if the number of subproblems is greater than the number of nodes in the hypercube, then we need to assign new subproblems to nodes 1, 3, 5, 7 after they become released after the first step.

A node is considered released if it completes its computation, communicates its result to its dimensional neighbor, and is free to accept another problem for computation.
2.2 Algorithms Related to Hypercubes

There is a significant amount of literature relating to algorithmic techniques for several parallel architectures [12]. A study of these algorithms reveals that the location of instructions and data among the computing nodes as well as the method of node synchronization are important for efficient algorithm implementation. Two classifications exist among contemporary parallel implementations. One is to have all the nodes execute the same set of instructions in lock-step read from a global memory and operate on separate sets of data relating to their subproblem domains. This technique is referred to as Single Instruction Multiple Data (SIMD) computation. This method, however, introduces access contention problem for the large shared memory. As the number of nodes is increased, this problem severely degrades performance. Another method which solves the contention problem, operates by having each node execute instructions from its own private memory. All the nodes execute the same instructions, but they do so asynchronously. Synchronization is achieved by having nodes wait for data from a companion node whenever a need arises. The companion node completes the data transaction when it is ready to communicate. This technique is referred to as Multiple Instruction Multiple Data (MIMD) computation.
Several authors have reported efficient matrix and graph algorithms on both MIMD and SIMD models with hypercube interconnection [4, 5, 14]. Since the goal of our study is incomplete hypercube of the message passing type which avoids shared memory contention problem for large configurations, in this study we implicitly refer to MIMD type hypercubes. Now let us study the algorithms for message passing, routing and broadcasting, adopting from Katseff's work.

2.2.1 Algorithms for Message Passing in MIMD Hypercubes

From the definition of the complete hypercube it is clear that the number of nodes $m$ in a hypercube is exponentially related to its dimension $d$. If we need to increase the power of the hypercube, we must invest in doubling the number of nodes for each increment in the dimension $d$. Assuming we have a hypercube of a given dimension, even if one node is down, we can't use the existing algorithms as they need the complete hypercube to run. It is clearly desirable from an economic and operational standpoint to keep the hypercube running even at a degraded performance level. Katseff in [10] proposed Incomplete Hypercube which has the potential to overcome this limitation. Before we proceed with the main discussion, let us investigate Katseff's proposal. Message passing architectures need simple and deadlock-free communication. Two fundamental communication primitives are necessary for message passing: routing and broadcasting. Routing is point-to-point communication while broadcasting is single source multiple destination communication. Based on the work of Sullivan and Bashkow for complete hypercubes [16], Katseff proved the existence of algorithms that can do routing and broadcasting for incomplete hypercubes. Let us look at these algorithms. We will adopt Katseff's definitions for this part of the discussion.

Let $src$ denote the message source, $dest$ denote the message destination, and $reladdr$ denote the relative address of two nodes computed as the bitwise Exclusive-OR of their node numbers. $\oplus$ denotes bitwise Exclusive-OR. A communication link has link number $i$ if it connects two nodes which are dimensional neighbors in
dimension \( i \). For example, the link between nodes 101 and 001 is link 2 because the two nodes differ in bit 2. Let us review the algorithms for routing in complete and incomplete hypercubes.

**Algorithm ROUTE_COMPLETE:**

```plaintext
/* Send or forward message from src to dest in complete hypercube [10] */

if (src == dest)
    Send message to local node
else
    reladdr = src ⊕ dest
    Start with the most significant bit of reladdr
    \( i \) = the first bit with value 1
    Send message on link \( i \)
endif
end /* ROUTE_COMPLETE */
```

In an incomplete hypercube, however, all the links indicated by link \( i \) above may not exist. So we need to ignore the non-existent links.

Figure 4 shows the message routing from 011 to 100 in a complete hypercube of \( m = 8 \) nodes. Figure 5 shows the same message routed in a incomplete hypercube of the same dimension.

Since the longest message path is no more than \( d = \log m \), and the algorithms reduce this number by 1 at each successful routing step, a message from any src to any dest is successfully routed. The formal proof that there is no deadlock is given in [10]. Now let us examine the broadcast algorithms. The following algorithm is designed for complete hypercubes.

**Algorithm BROADCAST_COMPLETE:**

```plaintext
/* Forward or originate a broadcast in a complete hypercube [16] */
```
Figure 4. Routing in Complete Hypercube.

Figure 5. Routing in Incomplete Hypercube.
Start at any node with weight set to $\log n$.

For each link $l$ from this node with $l < \text{weight}$:

send the message on link $l$ with a weight of $l$.

This algorithm sends the message to all the nodes in $\log m$ steps nonredundantly. That means, each node receives the message exactly once. However, introducing the earlier technique of ignoring missing links does not work from arbitrary nodes. For example, consider an incomplete hypercube with 3 nodes, 00, 01, 10. A broadcast sent from 01 with a weight of 2 reaches node 00, but has to go through 11 to reach 10 which is impossible. Hence, Katseff introduced an array $\text{travel}$. If a message is received with $\text{travel}[i]$ set to TRUE, the message should be sent on link $i$. We show the algorithms for complete and incomplete hypercubes below. These algorithms work from any arbitrary node.

Algorithm Modified.Broadcast.Complete:

```c
/* Forward or originate a broadcast in a complete hypercube [10] */

Start at any node with \text{travel} array filled to TRUE.

for each link $l$ from this node:

if $\text{travel}[i]$ = TRUE

send the message on link $l$ with array $\text{new}\text{travel}$ set as:

for $i = 0..(\log n - 1)$

if ($\text{travel}[i]$=TRUE AND $i < l$) $\text{new}\text{travel}[i]$ = TRUE .

endfor

endif

endfor

```

Once again, all the links implied by the condition $i < l$ may not exist in an incomplete hypercube. Hence we ignore the non-existent links in the case of incomplete hypercubes.
The proof works by showing that the route traced by broadcast is exactly the same route traced by the routing algorithm to all the individual nodes. The reader is referred to [10] for a formal proof.

Having established the existence of deadlock-free routing and broadcasting algorithms for incomplete hypercubes, we can proceed to study the architecture more thoroughly.

2.2.2 Fully Normal Algorithms (FNA)

One of our primary goals is to develop the concept of classes of algorithms that give generic solutions to a wide variety of problems. The first of these is what we refer to as Fully Normal Algorithms (FNA). Fully Normal Algorithms represent a class of algorithms based on hypercube architecture that use monotonous dimension collapsing to collect the result of parallel computations over the entire hypercube at node 0. Informally speaking, a typical FNA starts with the lowest (or highest) dimension and proceeds systematically towards highest (or lowest) dimension. As each dimension is considered, the nodes with the dimension bit set to 1 communicate their results to their dimensional neighbors that have their dimension bits set to 0. Released nodes may be allocated new subproblems from the ordered set of unsolved subproblems depending on the type of problem under consideration. We will return for an in-depth discussion on FNA. Presently let us look at two brief examples to get a feel for FNA.

2.3 Examples of FNA

Consider Figure 6. Each node reads and solves a subproblem that has the same index as the node number. For example node 0 reads subproblem 0, node 1 reads subproblem 1 etc. Now, all the eight nodes simultaneously solve their corresponding subproblems. Now, nodes with their $b_0$ set to 1 send their results to those with $b_0$ set to 0. In our example, nodes 1, 3, 5, 7 send their results to...
nodes 0, 2, 4, 6 respectively. Processors 0, 2, 4, 6 merge their local solutions with the received solutions. Now nodes with $b_1$ set to 1 send their solutions, if any, to their neighbors with $b_1$ set to 0. That means, nodes 2, 3, 6, 7 send to nodes 0, 1, 4, 5 respectively. Nodes 3 and 7 have nothing to send. So, 2 and 6 send to 0 and 4 respectively. That is, the subsolutions merged in the previous steps in 0, 2, 4, 6 got compressed into 0, 4. When we repeat the same operation in $b_2$, we can see that node 4 sends to node 0 and we have the subsolution sequence 0 — 8 in node 0. This is an example of static FNA. Processors that compute and send their solutions do not read any further subproblems for computation.

Now consider Figure 7. Here, initially we allocate subproblems to nodes in the node index order. So the first step of problem solving is same as in Example 1. But, now nodes 1, 3, 5, 7 are released and idling. Since we have more subproblems to solve than one per node, we need to reassign these four nodes for computation. In problems where the sequencing of solutions is important, the node reallocation is unclear at this point. By one technique that will be explained in later chapters, we allocate as shown in Figure 7 and follow the pattern of communications as in Example 1. It is easy to see how the dimension collapsing in this case also produces the correct sequence in node 0.
Until now we have identified the building block algorithms that we are going to study in ensuing chapters by showing their implementation on the incomplete hypercube architectures. But, since we have two different definitions of the incomplete hypercube architecture, we merge these two definitions in the next chapter by showing the transformation that proves \( CH(m) \) and \( IH(m) \) are equivalent.

Figure 7. Dynamic FNA.
CHAPTER III
INCOMPLETE AND COMPOSITE HYPERCUBES

Sequential numbering of incomplete hypercube nodes allows easier programming. This is due to the concept of flow of data from higher numbered nodes to lower numbered nodes. In case of composite hypercube, nodes are not sequentially numbered and therefore, this idea of data flow cannot be used. It appears that if we ignore the node addresses of $IH(m)$, then it looks structurally similar to $CH(m)$. That means, under some virtual node numbering scheme these two definitions can be identical.

In this section we develop an algorithm to renumber a composite hypercube $CH(m)$ such that it is mapped to an incomplete hypercube $IH(m)$. In this context we may view the renumbering problem as an embedding problem. We need to embed $CH(m)$ into an $m$-node incomplete hypercube. We map the nodes (edges) of $CH(m)$ to the nodes (corresponding edges) of the incomplete hypercube. In embedding terminology it can be called a “dilation-one-expansion-one” embedding.

3.1 Renumbering Composite Hypercubes

The basic idea is to start with the smallest complete hypercube in the $CH(m)$ and assign it sequential numbering. Let $m = 2^{d_1} + 2^{d_2} + \ldots + 2^{d_k}$. Let us call $NQ_{d_k}$ the sequential numbering of all nodes in the smallest complete hypercube, that is, $Q_{d_k}$. $Q_{d_k}$ is adjacent to nodes in $Q_{d_{k-1}}$ in exactly one dimension. However, $d_k$ can be much smaller than $d_{k-1}$. We extend the sequential numbering of $Q_{d_k}$ by $d_{k-1} - d_k$ dimensions. Now this extended hypercube $Q'_{d_{k-1}}$ is equal in size to $Q_{d_{k-1}}$ and sequential numbering is preserved, that is, all the existing nodes in $Q_{d_k}$ have the smallest sequential numbers. Now we assign $0, 1, \ldots, 2^{d_{k-1}} - 1$ to the nodes of
$Q_{d_{k-1}}$ and add $2^{d_{k-1}}$ to the node numbers of $Q'_{d_{k-1}}$ and we call it $NQ_{d_{k-1}}$. It is easy to see that this method can be applied to each successively larger complete hypercube.

Formally the algorithm is stated below, followed by an outline of the proof:

Algorithm RENUM:

1. Let $m = 2^{d_1} + 2^{d_2} + \ldots + 2^{d_h}$
   
   $NQ_{d_i} = \text{a sequential numbering of nodes}$
   
   $2^{d_1} + 2^{d_1+1} + \ldots + 2^{d_h}$
   
   $D_i = \text{the description of } d_i\text{-dimensional}$
   
   hypercube in $CH(m)$
   
   $NQ_{d_k} = \text{any sequential numbering of nodes}$
   
   of $Q_{d_k}$
   
   $d_i = d_{k-1}$

2. $i = k$

3. while ($i \neq 1$)

   $NQ'_{d_i} = \text{EXTEND}(NQ_{d_i}, D_i, d_i)$
   
   $NQ_{d_i} = \text{REFLECT}(NQ'_{d_i}, C_i)$
   
   where $C_i$ is connecting dimension to $Q_{d_{i-1}}$
   
   $D_i = D_{i-1}; d_i = d_{i-1}$
   
   $i = i - 1$

end while

EXTEND($Q_{d_i}, D_i, d_i$)

1. $Q'_{d_i} = \text{INV\_REFLECT}(Q_{d_i}, D_i)$

   /* Performed $d_{i-1} - d_i$ times due to $D_i$ */

2. return($Q'_{d_i}$)
**REFLECT\( (NQ'_d, C_i) \)**

1. Copy Hypercube\_Node\_Addresses and Complement \( b_{C_i} \)
2. Copy Generated\_Node\_Addresses
   - Append 0 to the left of copy
   - Append 1 to the left of original
3. return(New Address Sets)

**INV\_REFLECT\( (Q_d, D_i) \)**

1. For each Missing Dimension do /* \( D_i \) bit=* */
   (a) Copy Hypercube\_Node\_Addresses and Complement Dimension Bit
   (b) Copy Generated\_Node\_Addresses
      - Append 1 to the left of copy
      - Append 0 to the left of original
2. return(New Address Sets)

An example of this renumbering scheme and repeated application of procedures REFLECT and INV\_REFLECT is shown in Figure 8.

In Figure 8, the injured hypercube has five nodes. That means we must be able to identify two complete hypercubes of dimensions 2 and 0 connected in some dimension. It is easy to see that we do have two complete hypercubes of these respective dimensions made up of nodes 001, 011, 101, 111 and 110 connected in dimension 0. The dimensional description \( D_i \) for each of these hypercubes can be derived by looking at the bit patterns of the nodes. For the 2 dimensional hypercube \( Q_{d_2} \), we have \( D_i = **1 \) meaning that \( b_0 = 1 \) for the nodes. For the 0 dimensional hypercube \( Q_{d_0} \), we need to derive the \( D_i \) as **0 by comparison to its neighbor. Now we can apply algorithm RENUM. First we grow the \( Q_{d_0} \) in its two free dimensions 2 and 1. The order is unimportant, but we start from...
dimension 2 for a systematic coverage. Procedure EXTEND runs two cycles of INV.REFLECT. First cycle makes a copy of address 110 and inverts $b_2$ producing 010. Then it appends a 0 to the new address of 110 giving 0 and a 1 to the new address of 010 giving 1. The new addresses are shown on the right and the step number is shown inside a circle.

In the second cycle, repeating the INV.REFLECT on the two node addresses using $b_1$, it is easy to see the generation of the four addresses, and their new numbers on the right labelled EXTEND in the table. Since we have completely extended into the available dimensions given by $D_0$, now procedure REFLECT must be applied to connect this grown hypercube to the existing 2 dimensional hypercube. Procedure REFLECT does the same operations as procedure INV.REFLECT except that it appends a 1 to the left of the grown addresses, and a 0 to the left of existing hypercube for the right hand entries. Since procedure INV.REFLECT generates ghost nodes, they must be ignored in the final reckoning. Now that we have a fully grown hypercube of the highest dimension, we ignore ghost nodes 010, 000, 100 and read the renumbered addresses from the right hand entries of the table. It is clear that we have the required addresses 000 through 100 for making the incomplete hypercube, and the order of the nodes is correct.

Figure 8. Renumbering CH(5) to IH(5).
For formal discussion, let $EQ_d$ be the extended hypercube that has been assigned sequential numbers. $EQ_d$ contains all the nodes of $d_{i+1}, d_{i+2}, \ldots, d_k$ dimensional hypercubes. In addition, all existing nodes have been assigned lowest possible numbers. Now we need to combine $Q_d$ with $EQ_d$ to form $RQ_{d+1}$. Note that $d_i = d$. The basic idea in procedure REFLECT is to assign sequential numbers to the $Q_d$ based on the numbering of $EQ_{d+1}$. For each node $v$ in $EQ_d$, find its neighbor $v'$ in $Q_d$ and assign it the same number as that of $v$. Append a 0 as the leftmost bit in the number of $v'$ and append a 1 in the number of $v$. Note that now every node has a $d_i + 1$ bit number and all numbers are unique and node numbers in $Q_d$ are lowest and sequential, thus numbers of all existing nodes in the combined hypercube are lowest and sequential. We call this combined hypercube produced by the procedure REFLECT a reflected hypercube or $RQ_{d+1}$.

Let us define INV_REFLECT to be a procedure that also numbers like REFLECT except it appends 1 as the leftmost bit in the number of $v'$ and append a 0 in the number of $v$. Now procedure EXTEND can be explained in terms of procedure INV_REFLECT.

The basic idea behind EXTEND is to embed a smaller dimension hypercube which has been numbered into a larger dimension hypercube, so that the new hypercube may be combined with the next complete hypercube in the composite hypercube. Suppose $m = 2^{d_1} + 2^{d_2} + \ldots + 2^{d_k}$ and $d_1 > d_2 > \ldots > d_k$. Furthermore suppose that all nodes in $d_i$-dimensional hypercube to $d_k$-dimensional hypercube have been numbered and collectively called $RQ_d$. Now we need to number $d_{i-1}$-dimensional hypercube such that numbering of $RQ_d$ is not violated. We observe that $d_{i-1}$-dimension hypercube's description specifies its dimensions and addresses of its nodes. This information can be used to "grow" the $d_i$-dimensional hypercube, one dimension at a time. By "grow" we mean doubling in size. Let $j = d_{i-1} - d_i$. Note that $j$ is the number of times $RQ_d$ needs to grow in order to be equal in size to $d_i$-dimensional hypercube. Each time, we select a dimension
and use INV.REFLECT to get new numbering for a hypercube of one higher dimension.

**Lemma 1** Procedure EXTEND($RQ_{d_i}, D_i, d_i$) correctly generates a sequentially numbered $d_i-1$-dimensional hypercube, such that $NQ_{d_i}$ is preserved.

**Proof:** Let us denote the given sequence of addresses of complete subcube of dimension $d_i$ by $S_g = (0, 1, \ldots, 2^{d_i} - 1)$ and the sequence of its copy by $S_c$. Procedure merge calls INV.REFLECT $d_i$ times where $d_i = d_{i-1} - d_i$. Each time EXTEND copies $S_g$ to $S_c$. Then it appends a 1 to the left of binary representation of every element of $S_c$ and 0 to the left of members of $S_g$. That means all the addresses in the result sequence $S_r$ are continuous in the range $(0, 1, \ldots, 2^{d_i} + 1)$ which represents a complete hypercube of dimension $d_i + 1$. $S_r$ becomes $S_g$ for the next cycle of INV.REFLECT. So it is clear that at termination EXTEND would have generated a $S_r$ that has the sequence $(0, 1, \ldots, 2^{d_i+1} - 1)$ which is nothing but the sequence $(0, 1, \ldots, 2^{d_{i-1}} - 1)$. This is clearly the complete subcube with the dimension $d_{i-1}$. Since existing node addresses and the description $D_i$ are used in generating the new node addresses, $NQ_{d_i}$ is preserved. □

**Lemma 2** Procedure REFLECT($NQ'_{d_i}, C_i$) correctly generates a sequentially numbered $d_i-1$-dimensional hypercube, such that $NQ_{d_i}$ is preserved.

**Proof:** By using the above logic for dimension represented by address bit $C_i$ it is easy to prove this result. The fact that $S_g$ and $S_c$ have switched roles in append operation is immaterial for the above argument. □

We formally state and prove the following theorem.

**Theorem 3.1** Every composite hypercube $CH(m)$ is an incomplete hypercube $IH(m)$.

**Proof:** Proof follows immediately from the lemmas above and the fact that every composite hypercube $CH(m)$ can be viewed as consisting of two components: $d_1$-dimensional complete hypercube and $CH(m - 2^{d_1})$. By induction assume that
$CH(m - 2^{d_1})$ is already sequentially numbered. We use EXTEND to extend this numbering to that of $d_1$-dimensional hypercube. We then use REFLECT to obtain the numbering of entire $CH(m)$. □

The theorem implies that it is sufficient to develop algorithms for incomplete hypercubes. As a consequence, in the rest of this discussion, we use the notation $CH(m)$ and $IH(m)$ to mean a sequentially numbered composite hypercube. In other words the definition of composite hypercube affords no extra computational power in comparison to an incomplete hypercube. Have we gained anything from the recursive definition of the $CH(m)$? The answer is affirmative. As will be seen later in Chapter V on size/performance studies, it is essential that we compute the number of released nodes in a computation cycle. The recursive definition of $CH(m)$ readily gives a recursive formulation leading to a closed form solution.
CHAPTER IV

FULLY NORMAL ALGORITHMS

As we have seen until now, Katseff's [10] work covers the basic message passing and Tzeng [18] and other authors concentrate on embeddings, task scheduling etc. But no applications like graph algorithms have been attempted on incomplete hypercubes. Furthermore, our main objective is to develop algorithms on incomplete hypercube architecture that can be used as building blocks in other complex algorithms. This evolutionary approach is more economical than adapting individual algorithms running on hypercubes that most probably have the same basic structure. In this chapter, we attempt to identify classes of algorithms that follow the same basic structure and show that they can be elegantly implemented on incomplete architectures.

4.1 Fully Normal Algorithms on Hypercubes

Recall from Chapter II that Fully Normal Algorithms represent a class of algorithms based on hypercube architecture that use monotonous dimension collapsing to collect the result of parallel computations over the entire hypercube at node 0. For example, several graph problems like bipartiteness, fundamental cycles, bridges, connected components, and minimum cost spanning forests [4] use fully normal algorithms. Fully normal algorithms have also been used in other areas, such as routing and compaction in VLSI design and computational geometry [14]. In this section we present Fully Normal Algorithms (FNA) for complete and composite hypercubes. We need to cover the following four cases.

1a. Static FNA on complete hypercube
1b. Static FNA on composite hypercube

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
2a. Dynamic FNA on complete hypercube
2b. Dynamic FNA on composite hypercube

Let us introduce some terminology useful for presenting FNA in a formal way.

4.1.1 Static and Dynamic FNA for Complete Hypercubes

In this section, we give a brief overview of a FNA for complete hypercubes; this will serve as a basis for development of an FNA for incomplete hypercubes. Algorithm FNA_COMPLETE can be stated in generic form as follows:

Algorithm FNA_COMPLETE:

PROCESS \( i \) /* process \( i \) runs on node \( i \) of the hypercube */

1. If root-node(i.e node 0) then
   form sub-problems \( P_i, i = 1, K \)
2. Let \( d \) be the dimension of the Complete Hypercube
   Let \( N = 2^d \) be the number of nodes
   Let \( a = a_{d-1}a_{d-2}\ldots a_0 \) be the address of a node
3. If \( P_i \neq 0 \)
   ALLOCATE_ALL(\( P_i \))
   \( S_i = \text{SOLVE}(P_i) \)
endif
3. \( i = 0 \)
4. While \(((S_i \neq 0) \text{ and } (P_i \neq 0))\)
   \( S_{i+1} = \text{COLLAPSE}(d_i) \)
   \( S_{t_i} = \text{MERGE}(S_i, S_{i+1}) \)
   ALLOCATE(\( P_i, a_{i} = 1 \))
   \( S_{i} = \text{SOLVE}(P_i) \)
\[ i = i + 1 \mod d \]

Endwhile

Informally, FNA_COMPLETE works by dividing the problem \( P \) into several sub-problems \( K \) and allocating the sub-problems to the available nodes. (In practice, a global array of subproblems is broadcast to all the nodes as part of initialization by the Cube Manager. During computation, each node computes the index of its subproblem and reads it from this array.). This is done by ALLOCATE. In case 1a, \( K \leq m \), ALLOCATE inside the while loop is blank. In case 2a, the released processors in each dimension compute their subproblem indices on a dynamic basis. Then ALLOCATE inside the while loop plays a crucial role. MERGE will do whatever is necessary to merge the local data in a node with the received data, and SOLVE runs the local computations on the merged data. Any node terminates its repeat-until loop if either it does not have a subproblem to solve or it does not receive a subsolution from its dimensional neighbor. These two conditions basically reflect each node's role as sender or receiver of subproblems respectively. The important point is that the direction of process flow must be maintained. An example of the allocation process is shown in Figure 9.

**Remark 1** The direction of dimension collapsing does not alter the end result in a FNA. That is, the solution converges onto the node with address 0 irrespective of whether we proceed from dimension 0 through \( d - 1 \) or from \( d - 1 \) through 0. This is true because, in any dimension \( i \) irrespective of the direction of approach, data from node with address bit \( b_i = 1 \) is sent to node with address bit \( b_i = 0 \). Consequently, node 0 receives in all the dimensions.

Once the computation terminates, node 0 transmits the final result to the Cube Manager. This step is necessary, for Cube Manager alone is capable of communicating with I/O devices. The issue of providing the nodes with disk I/O is discussed by some researchers. But, as of today, distributed I/O resources are not available commercially.
Now, let us look at ALLOCATE to deal with case 2a. The formal specification is shown below.

ALLOCATE(step-no, node-no)

/*Pi is the last sub-problem assigned in step (step-no - 1)*/

1. If step-no = 0 then
   return P(node-no)
2. else
   A = ROTATE(node-no, step-no)
   offset = COMP(A, d-1)
   return P(l+offset)

ROTATE(a, j) is the right rotation function that rotates a to the right by j bits, i.e., \( \text{ROTATE}(a, j) = a_{j-1} \ldots a_1 a_0 a_{n-1} \ldots a_{j+1} a_j \). The rotation with binary addresses is accomplished by applying the same rotation function to all its nodes. It can be easily seen that all adjacencies are preserved under rotation. COMP(a, j) is the number obtained by complementing the \( j^{th} \) bit of a, i.e.,
\[ \text{COMP}(a,j) = a_{n-1}a_{n-2}...a_j, \ldots , a_1a_0. \] step-no is the dimension under consideration and node-no is the node's own binary address. Practical hypercube operating systems running on each node provide the capability to read the node's address. The reader can easily convince him/herself about the correctness of this scheme.

The following lemmas and theorem prove the correctness of this algorithm running on a Complete Hypercube \( Q_d \). Detailed proofs of these may be found in [14] in the context of an application, but without loss of generality.

**Lemma 3** Let problem \( P \) be divided into \( K \) sub-problems \( P_i, 1 \leq i \leq K \). Then, for \( K \leq 2^d \) the algorithm FNA.COMPLETE solves \( P \) in \( d \) steps.

**Lemma 4** Let \( R \) be the set of nodes released at any step of FNA.COMPLETE then the set \( R \) forms a \( d-1 \)-dimensional hypercube. Furthermore, there exists a node \( r \in R \), such that \( r \) is adjacent to node 0.

**Lemma 5** Assume that in a re-assignment step, \( 2^{d-1} \) sub-problems are assigned to the released subcube, then node \( r_s \), the root node of this subcube, will contain the solution to this set of sub-problems in \( d-1 \) steps.

**Theorem 4.2** Algorithm FNA.COMPLETE terminates in \( d + \lceil 2(K - 2^d)/2^d \rceil \) steps with solution at node 0.

**Remark 2** The algorithms that assume a complete hypercube configuration with unlimited memory per node [4] fall into the category of Static FNA. The available data get divided into almost equal partitions \( P/2^d \) and each partition is distributed to one computing node. In SIMD architecture, the global data are held in the shared memory. However, memory contention introduces a latency factor [5]. It must be emphasized that the issue of dynamic balancing of computational load, although important, is outside the scope of this study.

In the next section, we show the effect of loss of symmetry and regularity offered by a complete hypercube while addressing the problems associated with incomplete hypercubes.
4.2 FNA for Incomplete Hypercubes

In this section we develop fully normal algorithms for incomplete hypercube $IH(m)$. The results are valid for $CH(m)$. For static FNA, the solution is the same as for static FNA on complete hypercube architecture. In the case of dynamic FNA, however, two fundamental problems need to be resolved for the development of a FNA for incomplete hypercubes. One problem is the determination of the number of released nodes in a given step $i$ of the computation cycle. The number of nodes released by a complete hypercube in a computation step is fixed by the dimension and equals $2^{d-1}$. For incomplete hypercubes, however, this number varies depending upon the dimension and the number of nodes. The second problem is the relative ordering of released nodes in a computation step. The idea is to reorder the nodes, and assign them the next set of problems to be solved. This relative number is a function of the dimension of the computation step and the node number.

Remark 3 Precise problem sequencing is required only for solving problems with strict hierarchical structure like partial sums, or the Single Row Routing Problem [14]. In the case of Spanning Forest computation, problems can be allocated in any arbitrary order. However, we use the strict ordering, as long as there is no performance penalty, to expand the scope of the class of algorithms we can deal with.

We solve the two problems mentioned above in algorithm ALLOCATE which allocates a new problem to a released node. Let us assume that we have an $m$-node composite hypercube, $CH(m)$ and $m = 2^{d_1} + 2^{d_2} + \ldots + 2^{d_k}$. ALLOCATE for a $CH(m)$ is based on two arrays, RELEASED NODES (RN[step-no]), and DIMENSIONAL NEIGHBOR (DN[step-no,node-no]) to solve the problems mentioned above. These two arrays need to be computed only once at system initialization. The largest number assigned to a node, that is, $m - 1$ is broadcast to all the node of $CH(m)$s at the beginning of the algorithm.
In case of fully normal algorithms on a complete hypercube $Q_d$, that is, FNA.COMPLETE, $2^{d-1}$ nodes are released in each computation step. However, in the case of $C H(m)$, the number of nodes released is functionally dependent on the dimension $d_1 + 1$ and $m - 1$ and vary from 0 to $2^{d_1-1}$. Let $L$ be the list of addresses of nodes in $C H(m)$ and let $SUM(i) = \sum_{j=0}^{m-1} b_{ij}$ be the summation of $i^{th}$ bits of all the nodes.

**Lemma 6** Let $RN[i]$ be the number of nodes released in $i^{th}$ step of the computation cycle, then $RN[i] = SUM(i)$.

**Proof:** We compute the number of released nodes in $i^{th}$ step of the algorithm by counting nodes with $i^{th}$ address bit set to 1. In $i^{th}$ step, nodes which have their $i^{th}$ bit = 1 send their computed sub-solution to their dimensional neighbors. So these nodes, after sending, become free and are released. □

As stated earlier, the second problem that arises in reallocation is the determination of the relative ordering of nodes that form an incomplete hypercube. In the case of a complete hypercube, it can be done by rotating the current address right by $i$ bits and complementing the $d - 1^{th}$ bit. However, due to variable number of released nodes in each cycle caused by several missing nodes, this process can not be used in an incomplete hypercube. Our main observation here is that the re-numbering of nodes (for each dimension) can be done a priori for incomplete hypercube, for the structure of a $C H(m)$ for a given $m$ does not change.

An example of allocation scheme for composite hypercubes is shown in Figure 10.

Let $C H(m)$ be extended to a $d_1 + 1$-dimensional hypercube. Let $RL(i)$ be the list of released nodes in step $i$. Our algorithm first reorders all the nodes in $RL(i)$ as if all the nodes in $d_1 + 1$-dimensional hypercube are present. We remove all the nonexistent nodes from this list and refer to the order among the remaining nodes as the allocation order. If a node occupies $i^{th}$ position in this list, it will be assigned the $i^{th}$ sub-problem from the next sequence of subproblems to be solved.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
We compute this number for each node released in each dimension and store it in array DN[]. Thus DN[i][j] denotes the sequence-number of the jth node in the CH(m) released in the ith step.

**Lemma 7** The relative number of the jth node released in the ith step is DN[i][j].

**Proof:** In the ith step we compute the relative number of all the released nodes in the bounding complete hypercube using the rotate (ROTATE) and complement (COMP) functions. Now compute the new relative addresses by scanning this rearranged list of nodes and numbering them from 0 by skipping over the node addresses that fall above m - 1 for the incomplete hypercube under consideration. Since we consider all the nodes in the released hypercube and renumber only the nodes available in the incomplete hypercube, it is clear that the renumbering scheme will assign the proper number to the nodes in the released incomplete hypercube. □

Two important points in the algorithm FNA.INCOMPLETE need to be mentioned here. First, the function FIND_NEIGHBOR returns NULL in case the computed neighbor has address greater than the m - 1 value. Otherwise, it returns the node address. Second, step 5b for receiving node executes only if N is not a NULL meaning that a neighbor is present to complete the pairwise communication. These two steps are necessary to make sure that a receiving
node is never blocked expecting to receive a sub-solution when the corresponding sending node is absent. Similar controls are unnecessary for a sending node as it is never blocked when expecting to send a sub-solution.

The parallel algorithm in terms of processes running in parallel, can be stated as follows:

Algorithm FNA.INCOMPLETE:

PROCESS i /* process i runs on node i of the cube */

1. If root-node (i.e., node 0) then
   form sub-problems $P_i, i = 0, K$
2. step-no = 0.
3. $P_i = \text{ALLOCATE}(\text{step-no}, i)$
4. $S_i = \text{SOLVE}(P_i)$
5. Repeat
   (i) if (receiver for this step) then
      (a) $N = \text{FIND-NEIGHBOR}(\text{step-no}, i)$
      if (N)
         (b) RECEIVE($N, S_j$) /* receive $S_j$ from neighbor */
         (c) $S_i = \text{MERGE}(S_i, S_j)$
      endif
   else if (sender for this step) then
      (a) $N = \text{FIND-NEIGHBOR}(\text{step-no}, i)$
      (b) SEND($N, S_i$) /* send $S_i$ to neighbor */
      (c) $P_i = \text{ALLOCATE}(\text{step-no}, i)$
      (d) $S_i = \text{SOLVE}(P_i)$
   (ii) step-no = (step-no + 1) mod log N
6. Until (sub-solution received $S_j = \text{NULL}$) OR (new sub-problem $P_i = \text{NULL}$)
ALLOCATE(step-no,node-no)

/* $P_i$ is the last sub-problem assigned in step (step-no - 1)*/

1. If step-no = 0 then
   
   $I = N + 1$
   
   return $P_{(node-no)}$

2. else
   
   offset = DN(step-no,node-no)

   $P_{l'} = P_{(i+offset)}$

   $l = l + RN(step-no)$

   return $P_{l'}$

FIND-NEIGHBOR(step-no,node-no)

1. $N = COMP(node-no, step-no)$

2. If ($N < m$) return (N)

   else return (NULL)

In the next section we present the detailed comparative analysis of this algorithm.

4.3 Analysis of Algorithm FNA_INCOMPLETE

The major considerations in any parallel algorithm are the scheduling of the nodes and the communication overhead. No node should idle without a job at any time and the communication delay must be minimum among the set of nodes that need to communicate at any given stage of the algorithm. These criteria must hold...
good irrespective of the relative magnitude of the number of sub-problems and the number of nodes. Our primary aim in terms of communication delay is that two nodes that compute adjacent sub-problems must be dimensional neighbors on the hypercube. The allocation scheme described here meets these criteria while allocating sub-problems to nodes.

4.3.1 The Allocation Scheme

Assume that \( m \) is the number of nodes in the composite hypercube, \( d = d_1 + 1 \) is the dimension of the bounding complete hypercube, and \( N = 2^d \) is the number of nodes in the bounding complete hypercube. Let \( K \) be the number of sub-problems. Two cases arise in this scheme. For \( K \leq m \), the allocation is static in the sense that the sub-problems are allocated only once and the problem is solved in one computation cycle. For \( K > m \), however, all the sub-problems cannot be allocated to nodes and the computation proceeds in several cycles. In every dimension of the computation cycle we must allocate the remaining sub-problems to the released nodes in a pipelined fashion. Here the allocation scheme must address the problem of reallocation of released nodes. Let us examine these two cases in detail.

4.3.2 Allocation Scheme for \( K \leq m \)

In this no re-allocation is needed, as the number of nodes is sufficient to solve all the sub-problems after the initial allocation. Each node \( i \) is initially assigned sub-problem \( P_i \) which it solves using procedure SOLVE to obtain the sub-solution \( S_i \). In Step 0, each node with its 0th bit equal to 1 sends its sub-solution to its dimensional neighbor which contains the adjacent sub-solution. The neighbor is found by complementing the 0th bit of \( i \). For example, node 001 sends to node 000, and node 011 sends to node 010. In the same step, each receiving node \( i \) merges its sub-solution \( S_i \) with its received sub-solution \( S_{i+1} \) to obtain \( S_{i,i+1} \). In the next step this result is sent to the node that contains \( S_{i-2,i-1} \).
In the general step \( j < d_1 + 1 \), each node \( s = b_{d_1}b_{d_1-1} \ldots b_j \ldots b_1b_0 \) with \( b_j = 1 \) sends its sub-solution \( S(p,q) \) to node \( r = \text{COMP}(s,j) \), while each node \( r = b_{d_1}b_{d_1-1} \ldots b_j \ldots b_1b_0 \) with \( b_j = 0 \) receives a sub-solution \( S(p,q) \) from node \( s = \text{COMP}(r,j) \) and merges it with its sub-solution \( S(r,p-1) \) to obtain \( S(r,p) \). Similarly, in the step \( \lceil \log m \rceil - 1 \), which is the last step, node 0 receives sub-solution \( S(N/2,N-1) \) from the node \( N/2 \) and merges the received solution with its own sub-solution \( S(0,N/2-1) \) to obtain \( S(0,N-1) \). Because \( S(0,N-1) \) defines the solution to the entire problem, the algorithm terminates.

**Lemma 8** Let problem \( P \) be divided into \( K \) sub-problems \( P_i, 1 \leq i \leq K \). Then, for \( K \leq m \) the algorithm FNA.INCOMPLETE solves \( P \) in \( \lceil \log m \rceil \) steps.

**Proof:** The proof in this case follows from lemma 2 for complete hypercube since there is no reallocation of released nodes. However, since some of the nodes are missing, we need to show that the sub-solutions arrive in correct sequence at any node. This follows from the fact that all the sending nodes have higher addresses than their receiving nodes in any step and all the nodes in an Incomplete Hypercube are sequentially numbered from 0 to \( m - 1 \). Since there are sending nodes in all the dimensions 0 through \( \lceil \log m \rceil \), problem \( P \) is solved in \( \lceil \log m \rceil \) steps. □

**4.3.3 Allocation Scheme for \( K > m \)**

We now consider the case when the number of nodes is less than the number of sub-problems, i.e., \( m < K \). In this case, only \( m \) out of \( K \) sub-problems can be assigned to nodes in the first step, and we need to consider reallocation of released nodes. First we show in the case of a \( CH(m) \), the released nodes always form a composite hypercube that has a "local zero" node.

**Lemma 9** Let \( R \) be the set of nodes released in \( i^{th} \) step of the algorithm, then nodes in \( R \) form a Composite Hypercube \( CH(RN(i)) \). Furthermore, there exists a node \( r \in R \), such that \( r \) is adjacent to node zero of \( CH(m) \).
**Proof:** According to the definition of composite hypercube $CH(m)$, $m = \sum_{i=1}^{k} 2^{d_i}$, we can view it as a complete hypercube of dimension $d_1$ and a composite hypercube $CH(m - 2^{d_1})$. If the collapsing dimension is one which connects these two components, then $CH(m - 2^{d_1})$ is released. If the collapsing dimension is other than connecting dimension then the complete hypercube releases $2^{d_i-2}$ nodes. All nodes released by $CH(m - 2^{d-1})$ have dimensional neighbors in the complete hypercube and hence form a composite hypercube.

The nodes that are released in step $i$ contain $b_i = 1$. That means we need to generate all the combinations of $d$ bits keeping $b_i = 1$ to find all the released nodes. Quite obviously this set contains one combination that has $b_i = 1$ and all other bits zero. In a composite hypercube all the nodes generated out of these combinations may not exist. However, if any nodes in this particular dimension exist at all, then this node must be present as it has the least address among all the nodes in this dimension. This node is adjacent to node zero. We refer to this node as the 'released root' and denote it by $r_{si}$. □

Next we show that in the $CH(m')$ formed by $m'$ released nodes, the sub-solutions from the reallocated nodes converge onto their "local zero" node in $[m']$ steps.

**Lemma 10** Assume that in a re-assignment step $i$ in ALLOC, $m'$ sub-problems are assigned to a released subcube of dimension $d' = [m']$, then node $r_{si}$, the root node of this subcube, will contain the solution to this set of sub-problems in $d'$ steps.

**Proof:** The reassignment scheme in ALLOC results in a subcube of $m'$ "released" nodes which can be renumbered starting from 0 using $RN[i]$, and $DN[i][0\ldots m'-1]$ such that $r_{si}$ is labeled as 0. According to lemmas above this assignment will result in the solution $S_{(i,l+m')}$ being computed at node $r_{si}$ in $[m']$ steps. □

**Theorem 4.3** Algorithm FNA_INCOMPLETE for composite hypercube terminates successfully with the result in node zero.
Figure 11. Detailed Example of Node Allocation in CH(m).

**Proof:** After the initial set of assignments of problems \( P_0, P_1, \ldots, P_{m-1} \) in step 0, at each step new sub-problems are reassigned and the solution to each set of these sub-problems reaches the root node of their subcube \( r_{si} \) as per Lemma 10, and in another step merges with the existing solution at node zero of \( CH(m) \). □

The example in Figure 11 shows how the dimension collapsing and node reallocation work in a composite hypercube \( CH(m) \) with the details of the two global arrays \( RN[] \) and \( DN[][] \) that control the allocation order.

**Theorem 4.4** Algorithm *FNA_INCOMPLETE* is deadlock-free.

**Proof:** Two important features of the algorithm *FNA_INCOMPLETE* guarantee this condition. First, the function FIND_NEIGHBOR returns NULL in case the computed neighbor has address greater than the \( m - 1 \) value. Otherwise, it returns the node address. Second, in step 5b and 5c, the receiving node executes only if a neighbor in the executing dimension is present to complete the pairwise communication. These two steps make sure that a receiving node is never blocked expecting to receive a sub-solution when the corresponding sending node is absent. Sending nodes are never blocked as they always send their data to the nodes with lower numbers, and by definition of incomplete hypercubes, we know that all lower numbered nodes are present. □
So far we have shown the feasibility of implementing a generalized class of algorithms we call FNA on composite hypercubes. However, it is unclear at this point what size of the hypercube gives what kind of performance or more precisely speedup. Do we get a performance increment for every addition of a node? Since complete hypercubes have a simple linear size/performance tradeoff for FNA with exponential node increments, it is clear that we get something less than that for incomplete hypercubes of the intermediate size. How do they compare? In the following chapter, we investigate the size/performance tradeoffs involved in a composite hypercube with a given d-sequence and compare its performance in terms of nodes released in a computation cycle with a complete hypercube of dimension $d_1$ that includes an extended step of computation to compensate for the extra dimension available for the composite hypercube.
CHAPTER V

SIZE/PERFORMANCE TRADEOFFS FOR FNA

Complete hypercubes have a simple linear size/performance tradeoff for FNA. That is $Q_{d+1}$ has twice as many nodes as $Q_d$ and has twice the speedup. For example, $Q_4$ can solve 32 sub-problems as compared to 16 sub-problems solved by a $Q_3$ in 4 steps. This is due to the uniform and symmetric structure of the complete hypercubes. In case of incomplete hypercubes the speedup calculation is complicated by the fact that the number of nodes released in each step may be different. In this section, we compute the speedup that can be obtained by a $CH(m)$. It will be shown that over most of the range $2^d, 2^d+1, \ldots, 2^{d+1}$ incomplete hypercubes exhibit close to linear speedup, that is, as compared to $Q_d$. For example, we show that a $CH(1536)$ has a speedup of 1.45 as compared to a 1024 node complete hypercube. The ideal speed up would be 1.5. This means that $CH(48)$ is only 6% slower than an ideal 1536 node architecture.

5.1 Speedup of FNA

In order to facilitate the discussion, let us define speedup precisely. Speedup of an algorithm is usually defined as follows: Let the worst case complexity of the best-known sequential algorithm for a problem be denoted by $T_1$ and that of a parallel algorithm using $p$ nodes by $T_p$. The ratio $T_1/T_p$ is the speedup $S_p$ of the parallel algorithm.

With respect to fully normal algorithms, we define the speedup in terms of number of released nodes in a computation cycle, as the number of sub-problems solved is equal to the number of nodes released. Since we compare complete hypercubes with incomplete hypercubes, we define speedup achieved by a FNA on $CH(m)$ to be the ratio between number of nodes released by $CH(m)$ in $d_1+1$
The step computation cycle and the number of nodes released by $Q_{d_1}$ in $d_1 + 1$ steps. Where $m = 2^{d_1} + 2^{d_2} + \ldots + 2^{d_k}$. The $d_1 + 1$ step computation cycle must be considered for $Q_{d_1}$ as the number of steps in the computation cycle of $CH(m)$ is one more than that of $Q_{d_1}$.

In order to obtain an expression for the speedup, we need to compute the number of nodes released by a complete hypercube in a computation cycle. Let $R(m)$ be the number of nodes released by a $CH(m)$ in a computation cycle.

**Lemma 11** For a complete hypercube of dimension $d$, the number of released nodes in a computation cycle is given by $R(m) = d \cdot (m/2)$ where $m = 2^d$.

**Proof:** The proof follows from the fact that each dimension collapse releases $m/2$ nodes and there are $d$ dimensions. □

Thus speedup for $CH(m)$ may be defined as:

$$SP(m) = \frac{R(m)}{R(2^{d_1}) + \frac{2^{d_1}}{2}}$$

$$SP(m) = \frac{R(m)}{(d2^{d_1-1}) + \frac{2^{d_1}}{2}}$$

The second term in the denominator expresses the number of nodes released in the $d_1 + 1$ step of the computation cycle of $Q_{d_1}$. Before we compute speedup for the general case, it is illustrative to compute speedup for $CH(m)$ when $m = 2^d + 2^{d-1}$. This is a composite hypercube consisting of two complete hypercubes of successive sizes. Let us first compute the number of nodes released by $CH(m)$ in this case.

**Lemma 12** The number of nodes released in a computation cycle of $CH(m)$ is given by $R(m) = d \cdot 2^{d-1} + d - 1 \cdot 2^{d-2} + 2^{d-1}$ and $SP(m) = \frac{d+1}{2d+2}$ if $m = 2^d + 2^{d-1}$.

**Proof:** We may consider $CH(m)$ as composed of two complete hypercubes of dimensions $d$ and $d - 1$. During collapse of first $d - 1$ dimension both hypercubes release half of their nodes, independently. In the $d^{th}$ step, $Q_d$ releases half of its
nodes; however $Q_{d-1}$ does not have any nodes in this dimension and hence releases no nodes. Thus $Q_d$ releases $2^{d-1}$ nodes for $d$ steps, while $Q_{d-1}$ releases $2^{d-2}$ nodes for $d - 1$ steps. The third term in the expression represents the release of all the nodes of $Q_{d-1}$, while $d + 1$ dimension is collapsed.

The speedup in this case is computed as follows:

$$SP(m) = \frac{R(m)}{R(2^d) + \frac{2^d}{2}}$$

$$= \frac{d \cdot 2^{d-1} + (d - 1) \cdot 2^{d-2} + 2^{d-1}}{d \cdot 2^{d-1} + 2^{d-1}}$$

$$= \frac{d + \frac{d}{2} - \frac{1}{2} + 1}{d + 1}$$

$$= \frac{3d + 1}{2d + 2}$$

This completes the proof. □

Note that this type of hypercube is quite efficient and efficiency is better for larger sizes. For example, with $d = 5$, $SP(48) = 1.33$, while with $d = 10$, $SP(1536) = 1.45$. This shows that $SP(48)$ ($SP(1536)$) is only 11% (6%) slower than its idealized counterpart with the same number of nodes. Where, by ideal architecture we refer to an architecture with linear speedup.

In order to generalize the result in the lemma above to arbitrary $m$, we note that

**Theorem 5.5** The number of nodes released by a composite hypercube $CH(m)$ in a computation cycle is given by

$$R(m) = \sum_{i=1}^{k} d_i 2^{d_i-1} + \sum_{i=1}^{k-1} i 2^{d_{i+1}}.$$  

**Proof:** As per definition of a composite hypercube, one may consider a $CH(m)$ as consisting of two components: a $d_1$-dimensional complete hypercube and $CH(m - 2^{d_1})$. Let $r$ be the nodes in the non-complete part of $CH(m)$, that is $r = m - 2^{d_1}$. $CH(m)$ has a $d_1 + 1$ step computation cycle. In the first $d_1$ steps both components compute independently. It is clear that $d_1$-dimensional complete hypercubes will
release $2^{d_1-1}$ nodes in each cycle, thereby releasing a total of $d_1 \cdot 2^{d_1-1}$ nodes in $d_1$ steps. On the other hand, the number of nodes released by $CH(r)$ is given by $R(r)$, and may be computed recursively. In the $d_1 + 1$ step, the last dimension, namely $d_1 + 1$, is collapsed and all the nodes in $CH(r)$ are released. Therefore, the number of nodes released by $CH(m)$ can be expressed by the following recurrence relation:

$$R(m) = d_1 \cdot 2^{d_1-1} + R(r) + r$$

Note that $R(r) = d_2 \cdot 2^{d_2-1} + R(r - 2^{d_2}) + r - 2^{d_2}$, substituting we get

$$R(m) = d_1 \cdot 2^{d_1-1} + (d_2 \cdot 2^{d_2-1} + R(r - 2^{d_2}) + r - 2^{d_2}) + 2^{d_2} + 2^{d_3} + \ldots + 2^{d_k}$$

solving this recurrence in this fashion, we get the desired result. □

The speedup $SP(m)$ for $CH(m)$ can now be computed easily.

**Corollary 1** The speedup achieved by a composite hypercube $CH(m)$ is given by the following equation:

$$SP(m) = \frac{\sum_{i=1}^{k} d_i \cdot 2^{d_i-1} + \sum_{i=1}^{k-1} i \cdot 2^{d_i+1}}{(d_1 + 1)2^{d_1-1}}$$

Figure 12 shows the speedup of an FNA on composite hypercubes $CH(m), m = 32, 33, \ldots, 128$.

It may be seen from Figure 12 that not all composite hypercubes are useful. That is, some perform worse than their own (largest) component complete hypercubes. For example, $CH(33)$ has a speedup of 0.844, as compared to a 32-node complete hypercube. This implies there is degradation of effectiveness of the architecture by addition of a node. This is due to the fact that extra dimension added does not have enough nodes to justify the cost of computation in that dimension.

In this context, it is natural to consider the question of minimum number of nodes needed to be added to a complete hypercube so that composite hypercube performs better than the complete hypercube. Consider Table 1 listing the

---

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
speedup by \( CH(33) \) to \( CH(40) \). It can be seen from Table 1 that \( CH(39) \) is the smallest composite hypercube that performs equal to or better than a 32-node complete hypercube. For a given power of two, \( 2^d \), we call such a composite hypercube, a critical composite hypercube \( CCH(m') \) and refer to \( CN(2^d) = m' - 2^d \) as the critical number of nodes. Table 2 lists the value of critical composite hypercube and their respective critical number of nodes.

The last line in Table 2 shows the number of nodes (in terms of percentage) needed to reach criticality. It can be observed that the number of nodes needed decreases for larger hypercubes showing that criticality is easier to achieve for larger hypercubes.

Table 1

Criticality at CH(40)

<table>
<thead>
<tr>
<th>Nodes ( m )</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>( SP(m) )</td>
<td>1.000</td>
<td>0.844</td>
<td>0.865</td>
<td>0.885</td>
<td>0.917</td>
<td>0.938</td>
<td>0.969</td>
<td>1.000</td>
<td>1.042</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
5.2 Analysis of CCH

In Figure 13 we show that the fraction of nodes needed to reach the critical stage decreases hyperbolically with increasing dimension by curve fitting. Here we develop an analytical argument.

**Lemma 13** The number of excess nodes available for the incomplete hypercube $IH(m)$ when compared to the maximal complete hypercube of dimension $d_1$ is given by

$$m' = m - 2^{d_1} = 2^{d_2} + 2^{d_3} + \ldots + 2^{d_k}.$$  

**Proof:** Using the equivalent Composite hypercube representation, it is easy to see that

$$m = 2^{d_1} + 2^{d_2} + 2^{d_3} + \ldots + 2^{d_k}.$$  

Subtracting $2^{d_1}$ the result is obtained. □

**Lemma 14** The number of excess nodes released by $IH(m)$ in a computation cycle is given by

$$m'' = \sum_{i=2}^{k} d_i 2^{d_{i-1}} + \sum_{i=1}^{k-1} i 2^{d_{i+1}}.$$  

**Proof:** The number of released nodes in a computation cycle by a complete hypercube of dimension $d_1$ is $d_1 2^{d_1}$. Subtracting this from $R(m)$, the total number of released nodes, we get $m''$. □

**Table 2**  
Variation of CCH With Dimension

<table>
<thead>
<tr>
<th>Nodes $2^d$</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CCH(m')$</td>
<td>6</td>
<td>11</td>
<td>20</td>
<td>39</td>
<td>76</td>
<td>150</td>
<td>576</td>
<td>1143</td>
</tr>
<tr>
<td>$CN(2^d)$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>22</td>
<td>64</td>
<td>119</td>
</tr>
<tr>
<td>% Procs needed</td>
<td>50%</td>
<td>37%</td>
<td>25%</td>
<td>22%</td>
<td>19%</td>
<td>17%</td>
<td>13%</td>
<td>12%</td>
</tr>
</tbody>
</table>
To represent $m''$ in terms of $m'$, we need to simplify the corresponding expressions. We use the following fact from the degenerate nature of the corresponding $CH(m)$.

**Remark 4** Given the $d$-sequence of the $CH(m)$, it is easy to conclude that

$$d_2 \leq d_1 - 1, d_3 \leq d_2 - 1, \ldots, d_k \leq d_{k-1} - 1.$$

Notice that the equal condition is reached for any complete hypercube when only one node is absent. In other words, the value of $k$ is highest for a given complete hypercube when one and only one node is absent from the composite hypercube of next higher dimension.

That means $m'$ can be simplified to

$$m' \leq 2^{d_1} \sum_{i=1}^{k-1} \frac{1}{2^i} \leq \frac{2^{d_1}}{2^{k-1}} \sum_{i=1}^{k} 2^{k-i} \leq \frac{2^{d_1}}{2^{k-1}} (2^{k-1} - 1).$$

Now let us simplify $m''$. Expanding the summation series and combining like terms from both the series we get,

$$m'' = (1 + \frac{d_2}{2})2^{d_2} + (2 + \frac{d_3}{2})2^{d_3} + \ldots + (k - 1 + \frac{d_k}{2})2^{d_k}.$$
Using the fact stated above we can approximate this to get

\[ m'' = (1 + \frac{d_1 - 1}{2})2^{d_i - 1} + (2 + \frac{d_1 - 2}{2})2^{d_i - 2} + \ldots + (k - 1 + \frac{d_1 - (k - 1)}{2})2^{d_i - (k - 1)}. \]

Collecting like terms from the three additive terms over the entire series and reorganizing them around the summation series \( \sum_{i=1}^{k-1} i \), we get

\[ m'' = (\frac{1}{2} + \frac{2}{2^2} + \ldots + \frac{k - 1}{2^{k-1}})2^{d_i - 1} + (\frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{k-1}})d_12^{d_i - 1}. \]

By solving the two summation series in the brackets we get

\[ m'' = (2 - \frac{k + 1}{2^{k-1}})2^{d_i - 1} + (1 - \frac{1}{2^{k-1}})d_12^{d_i - 1} \]
\[ = 2^{d_i - 1}(2 - \frac{k + 1}{2^{k-1}} + d_1 - \frac{d_1}{2^{k-1}}). \]

**Theorem 5.6** The percentage of excess nodes to achieve criticality in \( IH(m) \) is approximately \( m'' = \frac{100}{d_i + 2} \).

**Proof:** The approximate value of \( m'' \) represented in terms of \( m' \) is given by

\[ \frac{m''}{m'} = \frac{2^{d_i - 1}2^{k-1}}{2^d(2^{k-1} - 1)}(2 + \frac{d_1 - k - 1}{2^{k-1}} + d_1) \]
\[ = \frac{1}{2(2^{k-1} - 1)}(2^k + (d_1 - k - 1) + d_12^{k-1}) \]
\[ = \frac{(d_1 + 2)2^{k-1}}{2(2^{k-1} - 1)}. \]

Even for moderately large values of \( d_1 \), the value of \( k \) would be large enough for the corresponding CCH to assume that \( \frac{2^{k-1}}{2^{k-1} - 1} = 1 \). Therefore the above equation can be simplified to

\[ \frac{m''}{m'} = \frac{d_1 + 2}{2}. \]
The condition for achieving criticality is $m'' > 2^{d_1 - 1}$. This means that the number of excess released nodes by the composite hypercube of dimension $d_1 + 1$ in a cycle of dimension collapsing must not be less than the number of excess released nodes by the complete hypercube of dimension $d_1$ in $d_1 + 1$ steps of dimension collapsing. It is obvious that if this condition does not hold, in several cycles of $d_1 + 1$-step dimension collapsing the composite hypercube progressively falls behind. Applying this condition for the minimum nodes needed we get

$$m' d_1 + 2 \geq \frac{2^{d_1}}{2}.$$ 

Adjusting the terms we get

$$\frac{m'}{2^{d_1}} \geq \frac{1}{d_1 + 2}.$$ 

Computing the percentage of excess nodes needed we get $m'' = \frac{100}{d_1 + 2}$. □

As the value of $d_1$ increases, this theoretical estimate gives a very good match to plotted values. Also, it is worthwhile to note that the speedup of composite hypercubes is very close to the ideal linear speedup architecture, after the critical number of nodes have been added as shown in Figure 12.

In this section the power of the recursive definition of $CH(m)$ enabled us to develop a closed form solution to the number of released nodes in a $CH(m)$. We also have shown, in this section, the existence of a critical size for the incomplete architecture below which computation is feasible but uneconomical for FNA. Since the speedup of critical composite hypercube grows closer to ideal linear speedup with increasing dimension, it indicates that we achieve close to ideal linear speedup for most of the range for FNA on incomplete hypercubes.

Now that we have established the computational advantages and limitations of incomplete hypercubes, let us investigate in the next chapter what kind of building block algorithms are feasible.
CHAPTER VI

GRAPH ALGORITHMS ON INCOMPLETE HYPERCUBES

In this chapter we show the implementation of several graph algorithms on incomplete hypercubes that fall into the category of static FNA. Then we show their dynamic FNA equivalents by relaxing the unlimited memory assumption. Graph algorithms, being highly general, are useful in many computationally intensive applications like VLSI, transportation networks, and other scientific and engineering fields. This kind of wide applicability of computationally intensive tasks makes it attractive to implement them on parallel architectures. In the following sections we show an implementation of the spanning forest algorithm which forms the basis for other graph algorithms like bipartiteness, fundamental cycle set, etc.

6.1 Spanning Forest on IH(m) With Unlimited Memory

In this section we develop an algorithm for the spanning forest problem on incomplete hypercube assuming the unlimited memory model. The Parallel Spanning Forest problem for complete hypercubes with unlimited memory per node is reported in [4]. Briefly, the idea works as follows:

In phase one, the \( m \) edges of the graph are distributed equally among the \( p \) nodes (i.e., the nodes \( p_k \), \( 0 \leq k \leq p-1 \), work on the subarray \( EDGE[k[m/p]...(k+1)[m/p] - 1, 0..1] \)). All the \( p \) nodes find spanning forests on their portion of edges in parallel using BFT [13, 17]. In the second phase these results are collapsed onto the node \( P_0 \) using monotonous dimension collapsing. This takes \( d \) steps where \( d = \log n \) is the dimension of the complete hypercube. At each step, a local spanning forest is computed using BFT at each of the receiving nodes. Finally the result is available in \( P_0 \).
The message flow is more complicated in the case of incomplete hypercube $IH(m)$ of $m$ nodes due to the absence of symmetry. Informally, a receiving node may be deadlocked if there is no sending node. A sending node may be blocked if there is no receiving node. Also, there is an additional problem of detecting such an absence on the run. From the definition of incomplete hypercube in [10], it is clear that all the nodes from 0 through $m - 1$ must be present. Recall that Katseff also showed the non-blocking nature of communication and broadcasting in an incomplete hypercube. Therefore we use the idea of broadcasting the address of the highest available node as part of the initialization so that every node can detect the presence or absence of its sending or receiving neighbor by comparing the neighbor’s address to $m$. Also, by restricting the communication among neighbors in any dimension $i$ from nodes with address bit $b_i = 1$ to nodes with $b_i = 0$, we are sure that no sending node is ever blocked. No communication is attempted to a receiving node that is absent because if a node is indeed absent, all its sending nodes must be absent as, by definition, they must have higher addresses than the receiving node.

The algorithm for incomplete hypercube with unlimited memory is formally stated below:

**Procedure Spanning_Forest_Incomplete1:**

begin /* process $i$ runs on node $i$ of the cube */

for all $P_k$, where $0 \leq k \leq p - 1$, do

    /* phase 1: */
    form an adjacency list $A$ of the edges $e_k[m/p], \ldots, e_{(k+1)[m/p]-1}$;
    use BFT on $A$ to form a spanning forest in LISTA;

    /* phase 2: $d$ steps of merging */
    for $0 \leq j < d$ do

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
if (k < 2^{d-j-1}) then
    if (address(sender) < m then
        recvw the spanning forest from P_{k+2^{d-j-1}};
    else
        send the spanning forest in LISTA to P_{k-2^{d-j-1}};
        exit;
    endif;
    form an adjacency list A of the edges in LISTA and LISTB;
    use BFT on A to form a spanning forest in LISTA;
endfor;
/* Po has the spanning forest of G, now */
endforall;

end; /* Spanning_Forest_Incomplete */

6.1.1 Analysis of Spanning_Forest_Incomplete

Here we informally prove the correctness of this algorithm and give its time complexity.

Theorem 6.7 The graph G' = (V, E') produced by Spanning_Forest_Incomplete at node P_0 of IH(m) is a spanning forest of the original graph G = (V, E).

Proof: We need to show that if any two vertices x and y are connected in the original graph G, then they remain connected in the new spanning forest G'. The proof follows from the fact that the set of edges forming the spanning forest at each node is a subset of edges derived from the given set of edges by elimination of non-forest edges. Since each selected edge of the final spanning forest traces a unique path from the node P_i to node P_0, and the fact that there is no blocked communication at any step in the algorithm proves that the vertices
connected by this edge in the original graph remain connected in the spanning forest. This argument can be applied to each selected edge of the spanning forest, thus completing the proof. □

The parallel time $T_p$ required by procedure Spanning_Forest With $p$ nodes in a complete hypercube of dimension $d$, is

$$ T_p = O(n \log p + (m_g/p)) $$

for a graph with $n$ vertices and $m_g$ edges. But in the case of incomplete hypercube, the dimension $d = \lfloor \log p \rfloor$ and hence the parallel time for a single cycle of dimension collapsing is

$$ T_p = O(n \lfloor \log p \rfloor + (m_g/p)) $$

This shows that the asymptotic time complexity is very close to that of the complete hypercube in the case of unlimited memory model of incomplete hypercube.

6.2 Spanning Forest on IH(m) With Limited Memory

In this section we investigate algorithm Spanning_Forest for an incomplete hypercube under the restricted memory model. We partition the $m_g$ edges of the graph into $K$ partitions. At each allocation, we allocate a data partition of $\lfloor m/K \rfloor$ to each free or released node in a pipelined fashion until we solve and communicate the results of all the $K$ partitions to node zero. The value of $K$ is calculated before starting the algorithm and depends upon the memory capacity of the individual node.

The formal algorithm is shown below:

**Procedure Spanning_Forest_Incomplete2:**

begin /* process $i$ runs on node $i$ of the cube */
for all $P_k$, where $0 \leq k \leq p - 1$, do

/* phase 1: */
form an adjacency list $A$ of the edges $e_k[m_2/K], \ldots, e(k+1)[m_2/K]-1$;
use BFT on $A$ to form a spanning forest in LISTA;

/* phase 2: */
$j = 0$; repeat
  if $(k < 2^{d-j-1})$ then
    if $(address(sender) < m$ then
      recv the spanning forest from $P_{k+2^{d-j-1}}$;
    else
      send the spanning forest in LISTA to $P_{k-2^{d-j-1}}$;
      $P_k = ALLOCATE(j, k)$;
      exit;
    endif;
  form an adjacency list $A$ of the edges in LISTA and LISTB;
  use BFT on $A$ to form a spanning forest in LISTA;
  $j = j \mod d$; 
  until $P_k$ is NULL;
  /* $P_0$ has the spanning forest of $G$, now */
endforall;

end; /* Spanning_Forest_Incomplete2 */
\[ l = N + 1 \]
\[ \text{return } P_{(\text{node-no})} \]

2. else
\[ \text{offset} = DN(\text{step-no, node-no}) \]
\[ P_i' = P(l+\text{offset}) \]
\[ l = l + RN(\text{step-no}) \]
\[ \text{return } P_i' \]

6.2.1 Analysis of Spanning_Forest_Incomplete2

This algorithm differs from the one that runs on complete hypercube shown in [4] in two places. The receiving node must check for the presence of its sending node before executing a receive-wait instruction. In the sending phase, each node that sends its computed result must be reallocated to solve a new problem. Procedure ALLOCATE allocates or reallocates a problem \( i \) to a node \( i \). The number of nodes released in a computation step is fixed for a complete hypercube of given dimension \( d \) and is equal to \( 2^{d-1} \). For incomplete hypercubes, this number varies depending on the dimension under consideration. This number is precomputed and stored in array RN[step-number]. We also need to precompute the relative ordering of released nodes in any dimension for reallocation. This figure is stored in array DN[step-number, node-number] so that any node can read its problem \( P_i \) knowing its node number and the current dimension under consideration. Sender’s or receiver’s address can be computed by complementing the address bit corresponding to the dimension under consideration.

**Theorem 6.8** The graph \( G' = (V, E') \) produced by Spanning_Forest_Incomplete at node \( P_0 \) of \( IH(m) \) is a spanning forest of the original graph \( G = (V, E) \).
Proof: The proof follows from the description given above and the fact that no sending node is ever blocked due to the absence of a receiving dimensional neighbor.

Computation of time complexity of Spanning_Forest_Incomplete is difficult as the number of processors is not a power of 2 and we may have several cycles of computation. Furthermore, it is of interest to find the relationship between the number of nodes and the speedup achieved by the algorithms. It follows from Theorem 4.5 that the speedup achieved by a composite hypercube $CH(m)$ is given by

$$SP(m) = \frac{\sum_{i=1}^{k} d_i 2^{d_i-1} + \sum_{i=1}^{k-1} i 2^{d_i+1}}{(d_1 + 1)2^{d_1-1}}.$$  

Now we present several algorithms with informal description of how they use algorithm Spanning_Forest_Incomplete to solve several fundamental graph problems.

6.3 Other Graph Algorithms for Incomplete Hypercubes

In this section, we develop algorithms for testing bipartiteness, finding fundamental cycle set, connected components and bridges of an undirected graph. We assume that the spanning forest is available at all the nodes of the hypercube for the algorithms presented in this section. This can be accomplished either by a simultaneous computation in all the nodes by exchange of intermediate lists rather than unidirectional transmission, or by a final broadcast of the spanning forest to all the nodes from $P_0$. Both of these techniques pose no problems for incomplete hypercubes and the asymptotic time complexity is unaffected.

6.3.1 Graph Bipartiteness

Since a bipartite graph is two colorable, each node checks this condition for its subset of edges after coloring the vertices of the spanning forest. Then several
steps of multi-cycle dimension collapsing and reallocation produces the final result in node $P_0$.

The formal algorithm adapted from [4] with modifications for incomplete hypercubes with limited node capacity is as follows:

function Bipartite_Incomplete : /* outputs Yes or No */
begin /* process $i$ runs on node $i$ of the cube */
    for all $P_k$, where $0 \leq k < p$, do
        SPANNING_FOREST_INCOMPLETE2;
        /* Produces spanning forest in LISTA */
        form an adjacency list $A$ of the edges in LISTA;
        use BFT on $A$ to assign odd and even colors to all vertices of $G$
        and store the colors into the array LEVEL;
        Bipartite = Yes;
        /* Initial computation in $p$ allocated nodes. */
        for $k[m_g/K] \leq i < (k + 1)[m_g/K]$ do
            if (LEVEL[EDGE[i,0]] = LEVEL[EDGE[i,1]]) then
                Bipartite = No;
                break;
            endif;
        endfor;
        /* Multi-cycle computation in reallocated nodes. */
        $j = 0$; repeat
            if ($k < 2^d-j-1$) then
                if (address(sender) < $m$ then
                    recvw the result from $P_{k+2^d-j-1}$ in TEMP;
                else
                    send the result in Bipartite to $P_{k-2^d-j-1}$
                $P_k = $ ALLOCATE($j$, $k$);
           endif;
        endfor;
end;
for $k[m/K] \leq i < (k + 1)[m/K]$ do
    if (LEVEL[EDGE[i,0]] = LEVEL[EDGE[i,1]]) then
        Bipartite = No;
        break;
    endif;
endfor;
exit;
endif;
Bipartite = Bipartite AND TEMP;
until $P_k$ is NULL;
endforall;
end; /* function Bipartite_Incomplete */

Predictably, the multi-cycle problem allocation is the main difference between the unlimited memory model and limited memory model. This must be done not only for the spanning forest, but for computing the boolean bipartite condition as well. Of course, the computation could be terminated as soon as any node finds that its subset of the problems shows the graph is not bipartite. The conditional execution of recieve-wait by the receiving node is necessary for incomplete hypercube. Since communication between neighbors is non-blocking, the result converges onto node zero. The speedup is still dominated by the spanning forest algorithm.

6.3.2 Fundamental Cycle Set

For this section, the graph is assumed to be connected. We form a spanning tree using procedure Spanning_Forest_Incomplete2 and make it available at every node. Since a spanning tree contains all the nodes of the graph, every edge in the corresponding co-tree forms a fundamental cycle (FC) with a subset of edges in
the spanning tree. Each node detects whether a given edge in its subset belongs to the co-tree, and if it does, goes on to find the corresponding FC using the spanning tree.

**Procedure Fundamental_Cycle_Incomplete:**

begin /* process $i$ runs on node $i$ of the cube */

for all $P_k$, where $0 \leq k < p$, do

SPANNING_FOREST_INCOMPLETE2;
/* Produces spanning forest in LISTA */
form an adjacency list $A$ of the edges in LISTA;

/* identify the local co-tree edges */
/* All the released nodes read and solve repeatedly; */
/* the exact technique can be imagined easily. */
for $0 \leq i < \lfloor m_g/K \rfloor$ do

$COTREE[i] = Yes$;

for $0 \leq i < n - 1$ do

if $k \lfloor m_g/K \rfloor \leq LISTA[i] < (k + 1) \lfloor m_g/K \rfloor$ then

$COTREE[LISTA[i] - k \lfloor m_g/K \rfloor] = No$;

/* find the fundamental cycles */
/* All the released nodes read and solve repeatedly; */
for $0 \leq i < \lfloor m_g/K \rfloor$ do

if ($COTREE[i] = Yes$)

then find the path between the vertices

$EDGE[k \lfloor m_g/K \rfloor + i, 0]$ and

$EDGE[k \lfloor m_g/K \rfloor + i, 1]$ using

BFT on adjacency list $A$;

/* the edge $k \lfloor m_g/K \rfloor + i$ and
the path form a Fundamental Cycle */
endif
endfor;
/* now, the set of $m_g - n + 1$ Fundamental Cycles are spread
over the cube */
endforall;
end; /* Fundamental_Cycle_Incomplete */

6.3.3 Connected Components and Bridges

Here also we assume the graph is connected. Since a bridge in a graph can
disconnect a graph into two sub-graphs, it must necessarily appear in every span­
n ing tree. Since, by definition, bridges can not form fundamental cycles, it is clear
that if we can remove the edges of a spanning tree that form fundamental cycles
with edges in its co-tree we can get the bridges. Corneil's sequential algorithm [3]
for finding bridges of a connected graph, forms a spanning tree and collapses into
supervertices all the vertices of the spanning tree belonging to a component of
the co-tree. The parallel algorithm due to [4] uses an array $ROOT[0..n - 1]$ at
each node to store the connected components. For a vertex $v$ $ROOT[v]$ stores
the lowest vertex number in its component. The procedure in terms of procedure
Spanning_Forest_Incomplete can be stated as follows:

Procedure Connected_Components_Incomplete :
begin
  for all $P_k$, where $0 \leq k < p$, do
    SPANNING_FOREST_INCOMPLETE2;
    form an adjacency list $A$ of the edges in the spanning forest;
    use BFT on $A$ to get the connected components in array $ROOT$;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
# of connected components = the number of distinct entries in ROOT;

endforall;

end; /* Connected_Components_Incomplete */

The algorithm for finding bridges is specified below. Note that each time a for loop is shown for a data partition \([m_g/K]\) it means that the full parallel algorithm for incomplete hypercubes with node reallocation is run.

An outline of the formal algorithm is as follows:

Procedure Bridges_Incomplete :
begin

for all \(P_k\), where \(0 \leq k < p\), do

1. SPANNING_FOREST_INCOMPLETE2;
   /* Here we have a spanning tree of G in TREE array. */

2. /* Each node identifies local co-tree edges
   for all the assigned problem partitions as
   shown under procedure Fundamental Cycles. */
   /* Each node updates COTREE array belonging
   to the subset of problems assigned to it. */

3. Get the connected components of the co-tree
   using Connected_Components_Incomplete.

4. Form Supervertices by collapsing these connected components
   and form a modified graph using the supervertices and
   the spanning tree. (Note that the resulting graph is
   fully connected.)

5. Categorize the set of edges available to each node
   in parallel as bridges or non-bridges
   using the condition that if a bridge is removed from
   the graph, it disconnects the graph into two components.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
endforall;

end; /* Bridges_Incomplete */
CHAPTER VII

CONCLUSIONS

7.1 Conclusions and Future Directions

We have shown the architectural fusion of two independent definitions of incomplete hypercube architectures, incomplete and composite hypercubes. We have also shown that the transformation is lossless. Then we have identified a major class of algorithmic technique we call fully normal algorithms used on complete hypercubes and have shown its computational feasibility on incomplete hypercube in both static and dynamic models of problem allocation. We proceeded to identify the existence of critical size of the incomplete architecture below which computation of FNA is feasible but uneconomical. We have postulated a new speedup definition necessary to deal with dynamic FNA and have shown that over most of the range of the architecture, the performance of incomplete hypercube is very close to that of the ideal linear speedup of complete hypercube. We have shown applications in graph theory by developing algorithms for incomplete hypercubes that can be used as building blocks in other computationally intensive complex algorithms.

We have provided generic algorithms on incomplete hypercube architecture that can be used in building more complex algorithms with minimal incremental effort.

The problem of detecting the presence of a maximal composite hypercube under random node failure is an open problem. The resolution of this problem coupled with the renumbering scheme provides for automatic graceful performance degradation under random node failure conditions. Finally problems that have
different structural properties than FNA need to be studied and classified for effective implementation. As technological advances make larger hypercubes possible, incomplete hypercubes become very important and useful.
BIBLIOGRAPHY


63

