12-1992

Pargal: A Package for Parallel Graph Algorithms

Winston Anand-Kumar

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/901
PARGAL: A PACKAGE FOR PARALLEL GRAPH ALGORITHMS

by

Winston Anand-Kumar

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
December 1992
PARGAL: A PACKAGE FOR PARALLEL GRAPH ALGORITHMS

Winston Anand-Kumar, M.S.
Western Michigan University, 1992

In this thesis, we have considered three graph problems namely, spanning forest problem, connected component problem and minimum cost spanning forest problem. Standard methods to solve these problems using sequential algorithms by various authors are presented for each problem. Efficient parallel algorithms for these problems have also been presented by various authors; however, most of these solutions demand a need for large number of processors, which immediately increases the cost of the hardware and in most of the cases it seems not very practical.

We present efficient solutions for the three problems on a hypercube parallel computer using \( p \) processors. We implemented our algorithm on the nCUBE-2 parallel computer and experimentally computed the speedup for each algorithm. Our experimental results are very encouraging in general and we observe superior results when compared to the best known algorithms.
ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Ajay Gupta for his encouragement and guidance during my master's program. He worked very close with me and brought the very best of me; without his help I would not have made so much progress. Secondly, I would like to thank the members of my committee, Dr. Fred Boals and Dr. Naveed A. Sherwani; I extend my sincere appreciation for their supervision and support. I would like to thank all the faculty and the staff in the Department of Computer Science who have ever provided opportunity, encouragement and support to me. Special thanks goes to Dr. Elise Donken for providing all the help on the nCUBE-2 and Dr. Donald Nelson for keeping the resource available to us all the time.

During my stay at Western Michigan University, I have met a lot of wonderful people, who have been a very good asset for me. I am always grateful for their support morally and academically. In particular I would like to thank Arun Shanbhag, Eruch Rustomji, Siddharth Bhingarde and Siva Kumar for their help in computer programming. However, I can not forget all my old friends from my country, especially Rizwanullah Sheriff and Santhanam, who have always been kind and given me all the encouragement that I need.

I am very grateful to my mother Bala Perinbaraj, my brother Einstien and my sister Annie for their love and support which never ends. Finally, I take this opportunity to thank my uncle Jaising George, without whose help, all my wishes would still be a dream.

Winston Anand-Kumar
To my Lord Jesus Christ for his blessings and to my father Dr. T. Perinbaraj, for his never ending love.
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.
Pargal: A package for parallel graph algorithms

Winston, Anand-Kumar, M.S.
Western Michigan University, 1992
# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** ................................................................. ii  
**LIST OF TABLES** ................................................................. v  
**LIST OF FIGURES** ............................................................... vi  

I. **INTRODUCTION** ............................................................... 1  
  1.1 Definitions and Notations ........................................ 4  
     1.1.1 Models of Parallel Computation ...................... 6  
  1.2 Preliminaries ............................................................... 11  
     1.2.1 Union and Find .............................................. 11  
     1.2.2 Hypercube Collapsing ................................. 15  
  1.3 Previous Work ........................................................... 17  
  1.4 Results ........................................................................ 19  

II. **PARGAL INTERFACE** ....................................................... 21  
  2.1 File ............................................................................. 22  
  2.2 Miscellaneous .......................................................... 22  
  2.3 Standard Graphs ....................................................... 23  
  2.4 Graph Algorithms .................................................... 23  
  2.5 User Algorithms ....................................................... 24  
  2.6 Input and Output ....................................................... 24  

III. **SPANNING FOREST** .......................................................... 26  
  3.1 An Efficient Hypercube Algorithm for SFP ................. 28  
  3.2 Theoretical Analysis .................................................. 33
Table of Contents – Continued

CHAPTER

3.3 Experimental Results .......................................................... 36

IV. MINIMUM COST SPANNING FOREST ........................................... 46

4.1 An Algorithm for MCSFP ....................................................... 49

4.2 Theoretical Analysis ............................................................. 51

4.3 Experimental Results ........................................................... 54

V. CONNECTED COMPONENTS .......................................................... 60

5.1 An Algorithm for CCP .......................................................... 63

5.2 Theoretical Analysis ............................................................. 65

5.3 Experimental Results ........................................................... 66

VI. CONCLUSION AND FUTURE RESEARCH ................................. 74

REFERENCES ................................................................................. 76
LIST OF TABLES

1. SF: $S_p$ With $V=250$ Using Our Algorithm ..................... 37
2. SF: $S_p$ With $V=250$ Using WS Algorithm ..................... 37
3. SF: $S_p$ With $V=700$ Using Our Algorithm ..................... 38
4. SF: $S_p$ With $V=700$ Using WS Algorithm ..................... 38
5. MCSF: $S_p$ With $V=250$ ............................................. 55
6. MCSF: $S_p$ With $V=700$ ............................................. 55
7. CC: $S_p$ With $V=250$ Using Our Algorithm ..................... 67
8. CC: $S_p$ With $V=250$ Using WS Algorithm ..................... 67
9. CC: $S_p$ With $V=700$ Using Our Algorithm ..................... 68
10. CC: $S_p$ With $V=700$ Using WS Algorithm ..................... 68
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>PRAM With Shared-Memory</td>
<td>8</td>
</tr>
<tr>
<td>2.</td>
<td>Algorithm UNION</td>
<td>12</td>
</tr>
<tr>
<td>3.</td>
<td>Algorithm FIND</td>
<td>13</td>
</tr>
<tr>
<td>4.</td>
<td>Hypercube Collapsing Technique for Processor $j$ of a Hypercube</td>
<td>14</td>
</tr>
<tr>
<td>5.</td>
<td>Hypercube Collapsing Using 3-Dimension Hypercube</td>
<td>16</td>
</tr>
<tr>
<td>6.</td>
<td>DFS Spanning Forest Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>7.</td>
<td>Parallel Spanning Forest Algorithm</td>
<td>31</td>
</tr>
<tr>
<td>8.</td>
<td>Spanning Forest Algorithm Using Union and Find</td>
<td>32</td>
</tr>
<tr>
<td>9.</td>
<td>SF:$S_p$ for $V=250$ Using Our Algorithm</td>
<td>41</td>
</tr>
<tr>
<td>10.</td>
<td>SF:$S_p$ for $V=250$ Using WS Algorithm</td>
<td>41</td>
</tr>
<tr>
<td>11.</td>
<td>SF:$S_p$ for $V=700$ Using Our Algorithm</td>
<td>42</td>
</tr>
<tr>
<td>12.</td>
<td>SF:$S_p$ for $V=700$ Using WS Algorithm</td>
<td>42</td>
</tr>
<tr>
<td>13.</td>
<td>SF:$S_p$ for $V=125$ Using Our Algorithm</td>
<td>43</td>
</tr>
<tr>
<td>14.</td>
<td>SF:$S_p$ for $V=125$ Using WS Algorithm</td>
<td>43</td>
</tr>
<tr>
<td>15.</td>
<td>SF:$S_p$ for $V=350$ Using Our Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>16.</td>
<td>SF:$S_p$ for $V=350$ Using WS Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>17.</td>
<td>SF:$S_p$ for $V=600$ Using Our Algorithm</td>
<td>45</td>
</tr>
<tr>
<td>18.</td>
<td>SF:$S_p$ for $V=600$ Using WS Algorithm</td>
<td>45</td>
</tr>
<tr>
<td>19.</td>
<td>Kruskal's Minimum Cost Spanning Forest Algorithm</td>
<td>48</td>
</tr>
<tr>
<td>20.</td>
<td>Parallel Minimum Cost Spanning Forest Algorithm</td>
<td>50</td>
</tr>
<tr>
<td>21.</td>
<td>MCSF:$S_p$ for $V=250$.</td>
<td>57</td>
</tr>
<tr>
<td>22.</td>
<td>MCSF:$S_p$ for $V=700$.</td>
<td>57</td>
</tr>
<tr>
<td>23.</td>
<td>MCSF:$S_p$ for $V=125$.</td>
<td>58</td>
</tr>
</tbody>
</table>
List of Figures – Continued

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.</td>
<td>MGSF:$S_p$ for $V=350$</td>
<td>58</td>
</tr>
<tr>
<td>25.</td>
<td>MGSF:$S_p$ for $V=600$</td>
<td>59</td>
</tr>
<tr>
<td>26.</td>
<td>DFS Connected Component Algorithm</td>
<td>61</td>
</tr>
<tr>
<td>27.</td>
<td>Parallel Connected Components Algorithm</td>
<td>64</td>
</tr>
<tr>
<td>28.</td>
<td>$CC:S_p$ for $V=250$ Using Our Algorithm</td>
<td>69</td>
</tr>
<tr>
<td>29.</td>
<td>$CC:S_p$ for $V=250$ Using WS Algorithm</td>
<td>69</td>
</tr>
<tr>
<td>30.</td>
<td>$CC:S_p$ for $V=700$ Using Our Algorithm</td>
<td>70</td>
</tr>
<tr>
<td>31.</td>
<td>$CC:S_p$ for $V=700$ Using WS Algorithm</td>
<td>70</td>
</tr>
<tr>
<td>32.</td>
<td>$CC:S_p$ for $V=125$ Using WS Algorithm</td>
<td>71</td>
</tr>
<tr>
<td>33.</td>
<td>$CC:S_p$ for $V=125$ Using Our Algorithm</td>
<td>71</td>
</tr>
<tr>
<td>34.</td>
<td>$CC:S_p$ for $V=350$ Using WS Algorithm</td>
<td>72</td>
</tr>
<tr>
<td>35.</td>
<td>$CC:S_p$ for $V=350$ Using Our Algorithm</td>
<td>72</td>
</tr>
<tr>
<td>36.</td>
<td>$CC:S_p$ for $V=600$ Using Our Algorithm</td>
<td>73</td>
</tr>
<tr>
<td>37.</td>
<td>$CC:S_p$ for $V=600$ Using WS Algorithm</td>
<td>73</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

High-performance computers are increasingly in demand in the areas of structural analysis, weather forecasting, fusion energy research, medical diagnosis, artificial intelligence, military defense, general engineering and many other scientific applications. Without superpower computers, many of these challenges to advance human civilization cannot be made within a reasonable period of time. Achieving high performance depends not only on using faster and more reliable hardware devices but also on major improvements in the processing techniques. In other words, coming up with better algorithms to solve the given problem efficiently.

High-performance computers are centered around the concept of parallel processors, that is, a computer with many processing units, or processors, which demands concurrent execution of many programs in the computer. Parallel-processing computers provide a cost-effective means to achieve high system performance through concurrent activities. The idea here is that if several operations are performed simultaneously, then the time taken by a computer can be significantly reduced. Given a problem to be solved, it is broken into a number of subproblems. All of these subproblems are now solved simultaneously, each on a different processor. The results are then combined to produce an answer to the original problem. This is a radical departure from the model of computation adapted during the past years in building computers—namely, the sequential uniprocessor machine.

Parallel computers have different architectures depending on the interconnection network used to connect the processors. Some of the well-known architectures are Linear Arrays, Meshes, Binary Trees, Butterfly networks and Hypercube
networks. The binary hypercube has emerged as one of the most popular network architectures for parallel machines. This is mainly due to its topological properties such as, diameter, recursive structure, efficient message routing and broadcasting, etc. Furthermore, many well-known architectures such as trees and meshes can be simulated by a hypercube very efficiently.

In order to better understand and utilize an existing parallel processing system, one must first identify the computational needs of important applications. Among the several applications which we have already discussed which needs high-performance computing, we have chosen Graph problems. Virtually, all areas of computer science uses graphs to organize data, to model algorithms, and its a powerful tool to represent computational concepts. Many branches of engineering and science rely on graphs for representing a wide variety of objects from electrical circuits, chemical compounds, and crystals to genetical processes, sociological structures, and economic systems. The same is true for operations research, where graphs play a crucial role in transportation and network flow problems. It is therefore important for these applications to develop efficient algorithms to manipulate graphs and answer questions about them in a shorter duration of time.

The main objective in this thesis is to implement various existing graph algorithms on the 128-processor nCUBE-2 [21] parallel processing machine for different structure and size of graphs. These algorithms solve fundamental problems that include connected components, breadth first search, depth first search, single source shortest paths, all pair shortest paths, traveling salesperson problem, spanning forest, biconnected components and minimum cost spanning forest. Through our literature search we found many existing parallel algorithms for these problems and chose the ones which are efficient in terms of time and space. We have
implemented different algorithms for the same type of problem in order to determine their performance on the NCUBE-2 and on the RK/CE (Reactive Kernal Cosmic Environment) System [28].

After narrowing our literature search and implementing various parallel graph algorithms, we came up with twenty different algorithms. We noticed that it will be very hard to study these graphs algorithms without having some common interface. Hence, we build a GUI (Graphic User Interface) which would allow us to draw our own graph and execute various parallel algorithm using our graph as the input. We used the Xview tool kit developed for X-Window system to build such an interface. We refer to this interface as the PARGAL (PARallel Graph ALgorithms) interface, details and features of PARGAL interface is presented in Chapter II.

Unlike the uniprocessor algorithms, where the performance is typically measured by their time and space requirements, we measure speedup (described in the later part) [19] in addition to time and space in the case of a parallel algorithm. When talking of space complexity in a multicomputer, we always emphasised on the amount of local memory each processor requires in order to solve a given problem. This is important since massive parallel computers today are built with small amount of local memory. We have experimentally computed the speedup for each algorithm on different size graphs. Studying our results we could understand the behavior of these algorithms, our objective was to find out which is superior compared to the other ones. We have also described our experimental results in detail at the end of each chapter after presenting our algorithms. Our results are very encouraging, which is not only viewed as a tool to find out the best algorithms but also to find out the areas where an existing algorithm could be improved in terms of speedup and other performance measures.

Our thesis is organized as follows. In the remaining of this chapter in Section 1.1 we present definitions and notations used in the rest of this thesis, Section
Chapter II exposes us to some of the techniques used in our algorithms to solve various graph problems, Section 1.3 summarizes the previous work and Section 1.4 ends with our results. We present the PARGAL interface in Chapter II which is used to draw different graphs and to execute various parallel algorithm. In Chapter III, Chapter IV and Chapter V we present our algorithms to solve spanning forest problem, minimum cost spanning forest problem and connected component problem respectively. Chapters III though V start with the general discussion of the problem and some conventional methods to solve these problems using sequential and parallel algorithms. Later, we present our algorithms, theoretical analysis for each algorithm and we end the chapter with the experimental results. Chapter VI summarizes our accomplishment and our vision for the future.

1.1 Definitions and Notations

A graph consists of a finite set of vertices and a finite set of edges connecting pairs of these vertices. A graph is directed when its edges have an orientation and thus provide a one-way connection. The notation \( G = (V, E) \) is used to represent a graph \( G \) whose vertex set is \( V \) and the edge set is \( E \). We denote the number of edges in \( G \) as \( m = |E| \) and the number of vertices in \( G \) as \( n = |V| \).

A matrix representation can be used for computer storage and manipulation of a graph. Let \( G \) be a graph whose vertex set is \( V = \{v_0, v_1, \ldots, v_{n-1}\} \). This graph can be uniquely represented by an \( n \times n \) adjacency matrix, \( A \), whose entries \( a_{ij} \), \( 0 \leq i, j \leq n - 1 \), are defined as follows:

\[
  a_{ij} = 1 \text{ if } \text{edge}(v_i, v_j) \in E \\
  a_{ij} = 0 \text{ if } \text{edge}(v_i, v_j) \notin E
\]

An adjacency matrix representation requires \( \Theta(n^2) \) space for an \( n \)-vertex graph. When each edge of a graph is associated with a real number, called its weight, the graph is said to be weighted. A weighted graph may be directed.
or undirected. The meaning of an edge's weight varies from one application to another; it may represent distance, cost, time, probability, and so on. A weighted matrix $W$ can be used to represent a weighted graph. Here, entry $w_{ij}$ of $W$ represents the weight of edge $(v_i, v_j)$. If $v_i$ and $v_j$ are not connected by an edge, the $w_{ij}$ may be equal to zero, or infinity or any appropriate value, depending on the application.

Another possible way to represent a graph $G$ is by using lists. The adjacency list for a vertex $v$ is a list of all vertices that are adjacent to $v$. Thus a graph could be represented by $n$ adjacency lists, one for each vertex. Notice that the adjacency list representation of a graph requires $\Theta(m + n)$ space while the adjacency matrix representation requires $\Theta(n^2)$ space. Hence, it is efficient to use adjacency list representation for graphs where $m < \Theta(n^2)$.

A path from an origin vertex $v_i$ to a destination vertex $v_j$ in a graph $G = (V, E)$, is a sequence of edges $(v_i, v_{i1}), (v_{i1}, v_{i2}), \ldots, (v_{ik}, v_j)$ from $E$, where $v_{ip} \neq v_{i_q}$ for $p \neq q$. The cost of path is defined to be the sum of the weights of the edges on the path. The shortest path problem is to find for every ordered pair of vertices $(v, u)$ a path from $v$ to $u$ with the smallest cost. A cycle is a path in which the origin and destination are the same. In an unweighted graph, the length of a path or a cycle is equal to the number of edges forming it.

A subgraph $G' = (V', E')$ of a graph $G = (V, E)$ is a graph such that $V' \subseteq V$ and $E' \subseteq E$. A connected subgraph is a subgraph $G' = (V', E')$ such that there exists a path between every pair of vertices $v_i$ and $v_j$, $v_i, v_j \in V'$ and $i \neq j$. A maximal connected subgraph is a connected subgraph $G'$ such that it is not a subgraph of any other connected subgraph of $G$.

If for every two vertices $u$ and $v$ in $V$, there is a path in $G$ joining $u$ and $v$, then $G$ is connected. Each connected maximal subgraph of $G$ is called a connected component $CC$ of $G$. A tree is a connected undirected graph with no cycles in it. A spanning tree of an unweighted graph $G = (V, E)$ is an undirected
tree \( T = (V, E'), E' \subseteq E \). A spanning forest \( SF \) of \( G \) is a set of undirected trees \( \{(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)\} = SF \) such that the \( V_i \)'s form a partition of \( V \) and each \( T_i \) is a (possibly empty) disjoint subset of \( E \). If \( G = (V, E) \) is an undirected weighted graph with costs on the edges. The problem of constructing a spanning forest \( SF \) with the least possible total cost is viewed as the minimum cost spanning forest problem, we refer to this problem as MCSFP and the spanning forest thus generated as MCSF.

Let \( T = (V', E') \) be a directed graph, \( T \) is said to have a root \( r \), if \( r \in V' \) and every vertex \( v \in V' \) is reachable from \( r \) via a directed path. If the underlying undirected graph of \( T \) is a tree, then \( T \) is a directed tree. If, moreover, the underlying graph of \( T \) is a subgraph of a connected undirected graph \( G = (V, E) \) and \( V' = V \), then \( T \) is a directed spanning tree in \( G \). A directed forest is a graph whose connected components are directed trees. If \( T \) is a directed forest such that each directed tree in \( T \) is a directed spanning tree of a component of an undirected graph \( G \) and vice versa, then \( T \) is called a directed spanning forest of \( G \). If the edges of \( T \) are all reversed, the resulting graph is called an inverted spanning forest of \( G \). Inverted spanning tree, inverted tree, inverted forest, etc. are defined similarly.

1.1.1 Models of Parallel Computation

Let us consider the random access machine (RAM) model for sequential computing before we present the parallel computation models. A RAM consists of a read-only input tape, a write-only output tape, a program, and a memory. The input tape consists of sequence of squares, where each square holds a positive integer or a negative integer. The tape head moves one square to the right whenever a read operation is performed. Similarly, when a write instruction is performed, an integer is printed on the write-only tape and the tape head moves one square to right.
The memory of RAM consists of a sequence of registers, \( r_0, r_1, \ldots, r_k \), without having any upper bound for the number of registers. Each register is capable of holding an integer of arbitrary size. The program has various instructions such as arithmetic instruction, input-output instruction, indirect addressing and branching instruction. The program is not stored in the RAM's memory and hence, we assume that the RAM cannot modify or change the program. All the computation takes place in a special register called the *accumulator* which is similar to other registers which can hold an arbitrary integer.

One of the popular model of a parallel computer is a Parallel Random Access Machine (PRAM) model [19] which is a generalization of RAM model. There are different types of PRAMs, however, they all have multiple independent processors and a single shared memory. An \( N \)-processor PRAM consist of \( N \) processors \( P_1, P_2, \ldots, P_N \) and a global shared memory as shown in Figure 1, each processor also has very small (usually constant) amount of memory. In an exclusive-read, exclusive-write (EREW) PRAM, the memory accesses of the \( N \) processors are constrained so that at most one processor can read from or write to any location in the shared memory at each time. In a concurrent-read, exclusive-write (CREW) PRAM, multiple processors are allowed to read from the same location in shared memory at the same time, but only one processor can write to any location during a given time. In a concurrent-write (CW) PRAM, multiple processors can write to single location in shared memory at the same time. Furthermore, PRAM's could be classified into SIMD (Single Instruction Stream, Multiple Data Stream) model and MIMD (Multiple Instruction Stream, Multiple Data Stream) model. PRAM models which execute same instruction sequence on every processor, with each instruction potentially affecting many data items simultaneously, falls into the SIMD model and PRAM models which execute multiple instruction sequence which affects many data items simultaneously, falls into MIMD model.
Figure 1. PRAM With Shared-Memory.

PRAM model is very attractive from a theoretical point of view, since a global shared memory is used. In practice, a global shared memory is an abstraction that is not easy to implement through hardware. This fact introduced different interconnected network architecture for multicomputers, some of the dominant architectures are hypercubes, trees, arrays and meshes. In these architecture, the global shared memory is distributed in equal-size blocks among the processors in the network, called local memory.

If a processor needs to access a desired location in the global memory, it needs to send a request to the other processor which has access to that memory. A processor responds to a request from another processor and the desired information is sent. From the programming standpoint, inter-processor communication is accomplished by using calls such as "send" and "receive". A processor could send information to another processor by giving the destination address and the data which needs to be sent to the other processor. The incoming data to a processor is stored in its input buffer. A processor could check its input buffer at regular
interval using "receive" call and process a request or it can use a call such as "re-
ceiveb" which would wait until it receives some information form other processors.
Information is transferred from one processor to another processor in the form of
packets through the network, usually the packet-routing is accomplished by some
packet-routing algorithms.

Binary hypercube has emerged to be one of the most popular network
architecture due to its topological properties, similar structure and efficient mes-
sage routing and broadcasting, we have implemented most of our work on the
NCUBE-2 which has this architecture. Hence, in the remaining part of this the-
sis a multicomputer will refer to an NCUBE-2 with a network architecture of an
hypercube. Here in this section we have defined a binary hypercube and other
network architecture for the benefit of the reader.

A $d$-dimensional hypercube $Q_d$ is defined as $Q_d = K_2 \times Q_{d-1}$, where $K_2$ is
the complete graph on two nodes, $Q_0$ is a trivial graph with one node and '$\times'$ is the
cross product operation on two graphs [12]. Each node of the hypercube can be
uniquely represented by a string of $\{0, 1\}^d$. Such a representation is called $address$
of the node. Let $u_{d-1}u_{d-2} \ldots u_0$ be the address of a node $i$ and let $w_{d-1}w_{d-2} \ldots w_0$
be the address of a node $j$ then the $hamming distance$ between two nodes $i$ and
$j$ is given by $\text{ham}(i, j) = \sum_{m=0}^{d-1} (u_m \oplus w_m)$, where $\oplus$ denotes the exclusive-OR
function. Hence, two nodes in $Q_d$ are connected if and only if their hamming
distance is 1. Thus, every node in $Q_d$ is connected to $d$ other nodes, one in each
dimension, which are called its $buddies$. For example if we look at a 3-dimension
hypercube shown in Figure 5, we notice that each processor is connected to 3
other processors, thus processor 0's buddy are 1, 2 and 4. Further more, processor
1 is processor 0's buddy in dimension 1, processor 2 is the buddy in dimension 2
and processor 4 is the buddy in dimension 3.

A $mesh network$ $M_{m \times n}$ with $mn$-processors has $m$ rows and $n$ columns
arranged into a 2-dimensional lattice. An $N \times N$ mesh of trees is constructed
from an $N \times N$ grid of processors by adding processors and wires to form a complete binary tree in each row and each column.

The RK/CE (Reactive Kernal Cosmic Environment) System simulates a hypercube multicomputer by using $N$ computers (typically workstations), where each computer is referred as an independent processor. Each processor (computer) executes instructions concurrently and coordinates its activities with other processors by sending and receiving messages. RK/CE system uses a distributed memory system, the memory at each processor is physically separate from other processors and is called the local memory. All the independent computers are assumed to be connected through a local area network (LAN), which allows messages to be passed between the computers. The advantage of using RK/CE system is that parallel algorithms can be developed and tested without having a parallel computer. For more details we refer the reader to [28].

In order to determine the effectiveness with which the multicomputer processors are being used, we have to measure the quantity speedup $S_p$ in addition to the time and space complexity of a regular sequential algorithm. Speedup, $S_p$ using $p$ processors is given by $S_p = \frac{t_0}{t_p}$, where $t_0$ is the time required to solve the given problem on a single processor using the best sequential algorithm and $t_p$ is the time of a parallel algorithm to solve the same problem using $p$ processors. Speedup is a quantity which measures the performance of an algorithm when it runs using more than one processor. Ideally if an algorithm takes $t_p$ time to run using $p$ processors, the same algorithm should take only $\frac{t_p}{2}$ time to run using $2 \times p$ processors since we have twice as many processors, however, this is not achieved in practice, in general.
1.2 Preliminaries

In this section we will be presenting some techniques namely Union and Find, which is used as a backbone for most of our algorithms and the hypercube collapsing technique which is used to communicate between different processors of the hypercube parallel computer in all of our algorithms.

1.2.1 Union and Find

In all of our algorithms disjoint Union and Find with path compression technique [1] is extensively used during the process of merging two spanning forests or to build a minimum cost spanning tree. This technique helped us in enhancing the speedup of our parallel algorithms by preserving the local spanning forests (discussed in Chapter III) build at each step.

Union and Find are two of the basic operations on a set, where Union($S_1$, $S_2$) replaces set $S_1$ by $S_1 \cup S_2$ assuming that $S_1$ and $S_2$ are disjoint. Find($a$), prints or returns the name of the set of which $a$ is currently a member, assuming that $a$ occurs in exactly one set.

We formally present algorithm Union and Find in Figure 2 and Figure 3, respectively. Let $a_1, a_2, \ldots, a_n$ be $n$ elements. Initially $a_i$ belongs to set $S_i$, $1 \leq i \leq n$. We use trees to represent sets. A set $S_i$ is represented by tree $T_i$ such that set $S_i$ is identified by the root $r_i$ of the tree $T_i$. Each tree has a set of nodes which basically represent the elements in the set and each node has a parent which points to another node. The root of the tree is the topmost node in the tree and the parent of the root points to 0. Union of two sets $S_1$ and $S_2$ is accomplished by simply merging the smaller set into larger set (i.e., change the parent pointer of the root of the smaller set's tree to point to the root of the larger set's tree). It is easy to see that Union thus takes $O(1)$ time.
Algorithm $\text{Union}$

**Input:** Sets $S_1, S_2$

(*Sets $S_1$ & $S_2$ are represented as trees $T_1$ & $T_2$ and identified by their roots $r_1$ & $r_2$ *)

**Output:** Set $S_1 = S_1 \cup S_2$

(*i.e., a new tree $T = T_1 \cup T_2$ *)

\[
\begin{align*}
&\text{if } |T_1| < |T_2| \text{ then} \\
&r_{1}.\text{parent} \leftarrow r_2 \\
&\text{else} \\
&r_{2}.\text{parent} \leftarrow r_1
\end{align*}
\]

end $\text{Union}$;

Figure 2. Algorithm UNION.
Algorithm \textit{Find} \\
\textbf{Input:} Sets $S$ and element $a$ \\
\textbf{Output:} Set name $s_a$ \\
\begin{verbatim}
  $u \leftarrow a$;  \\
  \textbf{while} $u.\text{parent} \neq 0$ \textbf{do}  \\
  \hspace{1em} $u \leftarrow u.\text{parent}$;  \\
  \textbf{end while}  \\
  root $\leftarrow u$;  \\
  (* root is the set name where element $a$ belongs *)  \\
  Let $P$ be the path from $a$ to root  \\
  \textbf{for} every node $v \in P$ \textbf{do}  \\
  \hspace{1em} $v.\text{parent} \leftarrow root$  \\
  (* perform path compression *)  \\
  \textbf{end for}  \\
  return root;  \\
\end{verbatim}  \\
\textbf{end \textit{Find};}  \\

Figure 3. Algorithm FIND.
Algorithm Hypercube_Collapsing

(* for every processor *)

begin

for \( i \leftarrow 1 \) to \( \log p \) do

Step 1: if the \( i^{th} \) bit of \( PE_j \) is 0 then
receive data from its buddy in the \( i^{th} \) dimension;

Step 2: if the \( i^{th} \) bit of \( PE_j \) is 1 then
send data to its buddy in the \( i^{th} \) dimension;

end for

end Hypercube_Collapsing

Figure 4. Hypercube Collapsing Technique for Processor \( j \) of a Hypercube.

In order to find the set to which an element \( a \) belongs, we simply traverse the path from \( a \) to the root and find the set name at the root. In addition we perform path compression, i.e., we now change the parent pointer of all the nodes that lie on the path from \( a \) to the root to point to the root. It is not hard to see that Find takes \( O(\log n) \) time if \( n \) nodes are in the tree. However, for a sequence of \( O(n) \) Unions and Find the time taken is \( O(n \alpha(n)) \) where \( \alpha(n) \) is defined to be smallest integer \( k \) such that \( A(k) \geq n \) and \( A \) is defined as:

\[
A(0) = 1,
\]

\[
A(i) = 2^{A(i-1)}, \text{ for } i > 0
\]

The function \( \alpha(n) \) grows extremely slowly. In fact, \( \alpha(n) \leq 5 \) for all "practical" values of \( n \), i.e., for all \( n \leq 2^{5536} \).
1.2.2 Hypercube Collapsing

Let us consider the hypercube collapsing technique which is repeatedly used in the rest of this thesis. Essentially speaking, hypercube collapsing technique is the communication path used to communicate data between processors in the hypercube parallel computer. The basic structure used for communication in our algorithms is given in Figure 4.

At each stage of this algorithm, a hypercube with dimension $d$ is reduced to dimension $d - 1$. Thus, for a hypercube with $p$ processors, it takes $\log p$ steps to compress the hypercube to its $0^{th}$ dimension (a single processor). As mentioned earlier, in the case of a parallel computer a larger task is divided into smaller tasks and solved independently at the same time using number of processors and the final solution is obtained by merging all the solutions from different processors. In the case of the hypercube computer, we initially distribute the tasks equally to all the processors. Each processor solves the given problem independently using the information in its local memory.

Once all the processors complete solving the given task they send or receive data based on the communication structure given in hypercube collapsing algorithm. There are $\log p$ stages in the algorithm. In stage $i$, every processor with $i^{th}$ bit 0 receives data from its buddy in the $i^{th}$ dimension. Note that the buddy in the $i^{th}$ dimension has $i^{th}$ bit as 1. For example, in a 3-dimension hypercube, each processor is immediately connected to 3 other processors, thus we say each processor has 3 buddies one in dimension 1, another in dimension 2 and yet another in dimension 3. In Step 2, we notice processor $PE_j$ sends data to its buddy if its $i^{th}$ bit is 1. Thus, at each step, half of the hypercube processors are sending data to the other half. Once a processor sends data to another processor it remains as a dormant processor.
Figure 5. Hypercube Collapsing Using 3-Dimension Hypercube.
In order to understand the above algorithm clearly, we have shown all the hypercube collapsing stages using a 3-dimension \((p = 8)\) hypercube in Figure 5. Initially, we have all the 8 processors active in Figure 5.a. In Stage 1 processor 1 sends data to processor 0, processor 3 sends data to processor 2 and so on as shown in the figure. The resulting 2-dimension hypercube is shown in Figure 5.b, where we see only the remaining 4 processors, since the other 4 processors are no longer active once they have sent their results. In the next stage processor 2 and 6 sends data to processor 0 and 4, respectively, reducing the hypercube from 2-dimension to 1-dimension. Thus we can notice that at each stage the dimension of the hypercube is reducing by one. Finally, in Stage 3, processor 4 sends data to processor 0. Thus, at each step the results are merged and the final result is obtained in processor 0 of the hypercube.

### 1.3 Previous Work

Efficient solutions for various graph problems using sequential algorithms were presented by various authors [1, 9, 16, 24, 31]. There are various sequential algorithms that exist to find solutions for graph problems such as minimum cost spanning forest, connected components and spanning forest. Some of the popular methods to find solutions for minimum cost spanning forest is by using Sollin's method [31], Prim's method [24], Kruskal's method [16] and Dijkstra's method [9]. Similar techniques could be used to find solution for spanning forest problem and connected component problem, some of the standard methods are by using breadth first search and depth first search [1].

Parallel algorithms for the above mentioned problems were presented by various authors [2, 4, 14, 20, 22, 27, 29, 34] and an extensive study of various parallel algorithms are presented in [19, 3]. Parallel algorithms to solve spanning
forest problems were presented using breadth first search technique, vertex col­

collapse technique, etc. Chin has proposed an $O(\log^2 n)$ time algorithm on a SIMD
using $\frac{n^2}{\log n}$ processors [33]. Dekel, Nassimi, and Sahni have developed an $O(\log^2 n)$
time hypercube algorithm using $\frac{n^2}{\log n}$ processors [7].

A good deal of work has been reported on parallel algorithms to find
the connected components of undirected graphs. These algorithms are based on
breadth first search, transitive closure and vertex collapse. Hischberg has pro­
pposed an $O(\log^2 n)$ time algorithm using vertex collapse technique on a SIMD
PRAM with shared memory and concurrent read using $n^2$ processors [13]. Savage
and Ja' Ja' have proposed an $O(\log^2 n)$ time algorithm using vertex collapse on a
SIMD PRAM with shared memory and concurrent read using $m + n \log n$ processors [29]. Kucera has proposed an algorithm using transitive closure on a SIMD
PRAM with shared memory and concurrent read and write using $n^4$ processors,
which takes $O(\log n)$ time [17]. In addition to these algorithms various authors
have presented algorithms using vertex collapse and other technique which are
described in [6, 11, 20, 22].

Parallel algorithms to find minimum spanning tree of a weighted, con­
nected, undirected graphs have focused on three classical sequential algorithms
namely Sollin’s method [31], Prim’s method [24], Kruskal’s method [16] and Di­
jkstra’s method [9]. Sollin’s algorithms is the best candidate, which could be
parallelized very easily. Various parallel algorithms have been presented using
Sollin’s method [8, 11, 22, 29, 34]. Kucera has presented a $O(\log m)$ time al­
gorithm using Kruskal’s method on a SIMD PRAM with shared memory and
concurrent read and write using $mn^4$ processors [17]. Yoo has presented an $O(m)$
time algorithm using Kruskal’s method on a MIMD tightly coupled PRAM using
$O(m)$ processors [35].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
1.4 Results

We implemented various parallel algorithms for some of the above mentioned problems and chose the ones which are efficient in terms of time and speedup $S_p$. We found Woo's and Sahni's algorithm very interesting because they were designed to run on an hypercube parallel computer using $p$ processors. Their algorithm gave good speedup and does not require huge number of processors in order to solve the problem efficiently. We found that their algorithm works good for dense graphs, we improved their algorithm to solve connected component problem and spanning forest problem. Our algorithm gives good speedup for all the graphs ranging from sparse to dense. We have also presented an efficient algorithm to solve minimum cost spanning tree. At the end of each chapter, we have presented our experimental results.

In general speedup is one of the measuring quantity in addition to time and space which measures the efficiency of a parallel algorithm when it runs using more than one processor. For each algorithm presented we have theoretically computed the speedup and analyzed the speedup relation by varying the number of edges $m$ and the number of vertices $n$. The behavior of the speedup for all the three algorithms were very much similar when $n = m$ and when $O(n^2) > m > n$.

In general we observed when $n = m$ (sparse graphs) the speedup does not increase significantly as we increase $p$ the number of processors. However, for minimum cost spanning forest problem, we observed that if the value of $m$ is considerably larger than $p$, the speedup does increase, irrespective of whether the graph is sparse or dense. When $m >> n$, we notice the speedup increasing if the number of edges $m$ is considerably larger than $p$. If $m \leq p$ we do not observe any speedup due to processor starvation. This is because the work load ($\frac{m}{p}$ edges) at each processor is very less. However, for all the three problems, we noticed a significant increase in the speedup by keeping the number of processors constant.
and increasing the number of edges, in other words by increasing the work load at each processor.

We used graphs of size 25 vertices to 700 vertices as the input for our algorithms, each graph ranges from sparse to dense by varying the number of edges randomly. All the algorithms were tested using same graphs and under similar conditions on the NCUBE-2 parallel computer. We consider only the computation time of each algorithms in order to calculate speedup, since communication time is same for all the problems.

The results obtained using our algorithm are very encouraging. We attain a maximum speedup of 83 using 128 processors to solve connected component and spanning forest problem for input graph size of 700 vertices and 242180 edges while Woo's and Sahni's algorithm gives a speedup of 26 for the same set of conditions. Notices, we get similar speedup for spanning forest problem and connected component problem because the structure of both the algorithms are the same. We get a maximum speedup of 32 for minimum cost spanning forest problem for the graphs with 700 vertices and 242180 edges using 128 processor. We are encouraged by the speedup achieved using our minimum cost spanning forest algorithm, since this algorithms could be very easily adapted to solve single source shortest path problems.
CHAPTER II

PARAGAL INTERFACE

As a part of our work we had developed several algorithms on both the RK/CE(Reactive Kernal Cosmic Environment) System [28] and on the nCUBE-2 [21]. We felt that it should be easy for the user to access these programs and also make it interactive so that the users can draw their own graphs and see the results on the screen graphically. Hence, we built a graphic interface on the SUN system using Xview tool kit developed for XWindows. Currently PARGAL is running on the Sun SPARC 1+ station using RK/CE system, the NCUBE version of PARGAL will be installed once we have the required file systems on the NCUBE. Here, in this section we will describe the different feature of this interface and how to use this with ease in order to get work done efficiently.

Since user-friendly is a very commonly used word these days, we have also tried to make this package as friendly as possible. We have two screens in the main window, one for drawing the input graph for the program and the other for displaying the messages for the user. The messages continuously inform the user of the status.

The package has five main menus, namely File menu, Miscellaneous menu, DrawGraph menu, Main Algorithms menu and User Algorithms menu. Under each menu title we have different options that will be discussed in detail later. Other than these menu buttons, we have a File Name bar which displays the current file, the default name displayed is GRAPH1. The user can change this name in order to open a new graph file or to store the current graph file with a different name. We also have a scroll bar to specify the dimension of the hypercube required to run an algorithm. The default value for the scroll bar is 0-dimension.
(one processor), the value in the scroll bar ranges from 0 to 7, however, currently we can use only a maximum of 3-dimension hypercube to run an algorithm on RK system since our installation of RK system allows a maximum 8-processors ghost cube.

There are two choice bars for selecting the operations for the vertex and edge in a graph. Under vertex, we have options to add a new vertex or delete an existing vertex, in the case of an edge, we have options to add an edge, delete an edge or change weight of an edge. When using the package, we are required to use only the left and middle mouse buttons. The left mouse button is exclusively designed to manipulate any thing to do with the vertex and the middle mouse button is designed to manipulate the edges. This will reduce the work of switching continuously between the choice bar and the graph screen.

2.1 File

This menu is mainly useful for manipulating files. Under this menu, we have options to open a file, save a file, save a file with a new name and to quit the program. We can see the name of the current active file on the File bar which appears on the top right corner of the window. When we select open or save option in the file menu, the programs gets the file name from the file bar. If we want to open a different file we should change the name on the File bar by clicking the left mouse button once on the File bar and then start typing the new name. If we want to save the file with a new name we can select the “Save As” option, which will pop up a window asking for a new file name.

2.2 Miscellaneous

We have three options in this menu, namely, “Clear Screen,” “Redraw Graph,” “Redraw Result.” “Clear Screen” option is used to clear the current
graph which appears on the canvas, “Redraw Graph” option is like refreshing, when deleting some vertices or edges and if we notice that the graph is not clear, we can use this option to redraw the graph. “Redraw Result” option is useful when we want to toggle between the original graph and the result obtained using a particular algorithm. This option will always draw the result of the last executed algorithm.

2.3 Standard Graphs

This menu is useful if we want to draw some standard graphs such as random graphs, complete graphs, complete Bipartite graphs, circuit, star, wheel and binary trees. When one of these options is chosen, one sees a window pop up asking for the number of vertices in the graph. A maximum of 30 vertices can be specified (for a clean and clear drawing on the canvas) and we have the option to continue generating this graph or to cancel the operation.

2.4 Graph Algorithms

This is the main part of this package, under this we have a list of graph algorithms which can be run on the graph we have created or input from a file. Some of the algorithms are Minimum Cost Spanning Tree, Depth First Spanning Tree, Breath First Spanning Tree, Connected Component, Bi-Connected Components, Shortest Path Problems, Traveling Sales Person Problem, etc., we will continue to add more algorithms as they become available. We may also notice the same type of algorithm appearing more than once, in such a case, we have also mentioned the technique used to build that algorithm, some techniques may be faster than other and in some we might not be able to notice the change. Once we have started an algorithm it might take a while to run the algorithms, as they are not part of the main package, it has to run a sub-shell in order to execute the
algorithm. Once the algorithm finishes, the new result will be displayed on the screen. If we want to see the original graph again we can go to the "Miscellaneous" menu and redraw the original graph.

2.5 User Algorithms

The idea of this option is to give more flexibility to the user, the user can develop his own parallel algorithms and add it to the procedure by giving its location in the user-file. In order to use this option the user must be aware of the input and output format used in this package. The user's program should read from our input and it should produce a compatible output so that our package can read the result and display it graphically. Currently, we are supporting three users algorithm, which can be changed if need arises.

2.6 Input and Output

The input format for most of our algorithms such as connected components, spanning forest, minimum cost spanning tree, breadth first search, bi-connected components are the same. The first line of the input file starts with an integer, the number of vertices $n$ in each graph, following that we have $n \times n$ integers, where each integer represents the weight of an edges, two vertices are not connected if the weight of the edges between them is zero. For problems such as single source shortest path we have the source vertex in the last line following the $n \times n$ integers in addition to the above format. Thus, a user needs to be aware of this format if they are going to develop parallel algorithms as part of the "User Algorithms" in the future.

PARGAL interface expects a standard output format so that the interface could read the output and display the results graphically on the screen. For problems such as connected component depth first search, breadth first search and
bi-connected component the output file should start with the number of vertices \( n \) in the graph, which is immediately followed by \( n \) lines, each line containing two integers. The first integer is the vertex number and second integer could be the order number in which it was searched in the case of searching algorithms, or the connected component number to which a vertex belongs in the case of connected components algorithm or a positive 1 to indicate that the node is an articulation node or negative 1 otherwise in the case of bi-connected component algorithms.

In the case of spanning forest algorithm and minimum cost spanning forest algorithm, the output format is the same as its input format. For single source shortest path problem, we have \( n - 1 \) lines, each starts with the source vertex then the destination vertex, the minimum cost to reach the destination and followed by this we have the path continuing till the end of line. For all pairs shortest path problem we have \( n \times n \) lines with each line having the same format as described for single source shortest path problem.

The name of the input file created by the interface in the area where it was installed is PARGAL.IN and the output file is PARGAL.OUT. Hence, the user should read the input file PARGAL.IN in their program and write the result to file PARGAL.OUT which would be read by the interface for the purpose of displaying the results.
CHAPTER III

SPANNING FOREST

A spanning tree of an unweighted graph \( G = (V, E) \) is an undirected tree \( ST = (V, T) \). A spanning forest \( SF \) of \( G \) is a set of undirected trees \( \{(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)\} = SF \) such that the \( V_i \)'s form a partition of \( V \) and each \( T_i \) is a (possibly empty) disjoint subset of \( E \). Furthermore, there does not exist an edge in \( E \) which connects a vertex \( u \in V_i \) and \( v \in V_j \) for some \( i \neq j \). Given a graph \( G \), one of the important problems in graph theory is to efficiently find the spanning forest \( SF \), we will refer to this problem as SFP. There exist several sequential [1] and parallel [7, 33, 34] algorithms to find a solution for the SFP.

Before considering the parallel algorithms for solving SFP, let us briefly go over a simple but efficient sequential algorithm to find spanning forest \( SF \) using depths first search [1]. This algorithm will be referred as the DFS spanning forest algorithm. Initially, all the vertices in \( V \) are marked "new" and the spanning forest is empty, then the algorithm shown in Figure 6 is executed. We start with any vertex \( v \in V \) and mark it "old." We next pick an adjacent vertex \( u \) of \( v \) which is originally marked "new" and add the edge \( (v, u) \in E \) to the spanning forest \( SF \). Next, we start the search from \( u \) until all the remaining vertices, that have a path from \( u \), have been marked "old." We again start the search from a vertex which is marked "new" and repeat the process.

It is easy to see that the DFS spanning forest algorithm takes \( O(n + m) \) time since every edge is considered a constant number of times and has a space complexity of \( O(n + m) \) if we use an adjacency list representation for \( G \). There also exist similar algorithms to solve SFP sequentially using breadth first search and the resulting spanning forested is called as the breadth first spanning forest.
Procedure $DFS(v)$

Input: Graph $G(V, E)$

Output: Spanning forest $SF(V, T)$

mark $v$ "old"

for each vertex $u$ adjacent to $v$ ido

if $u$ is marked "new" then

    add $(v, u)$ to $SF$;

    $DFS(u)$

endif

endfor

end $DFS$

Figure 6. DFS Spanning Forest Algorithm.
Let us now consider some of the parallel algorithms for SFP which have been developed for various parallel computation models. An $O(\log^2 n)$ time algorithm using $\frac{n^2}{\log n}$ processors on a CREW-PRAM was presented by Tsin and Chin [33]. The authors first find an inverted spanning forest from which a spanning forest is generated. Their algorithm finds a number of 1-tree-loop's [14], which is a directed graph whose vertices are called the supervertices (a supervertex is a vertex in $G$ of a 1-tree-loop). These directed edges of 1-tree-loop's contribute to the spanning forest, where the directions are ignored by the algorithm.

Dekel, Nassimi, and Salmi have developed an $O(\log^2 n)$ time algorithm to find a spanning forest of an $n$ vertex graph [7] using $n^3/\log n$ processors on an MIMD hypercube parallel computer. This algorithm was improved by Woo and Sahni for $p$ processors [32]. We refer to this algorithm as the WS algorithm. WS algorithm partitions the adjacency matrix of the graph $G$ in various ways and distribute these partitions among the processors of the hypercube. The algorithm use a depth first search or a breadth first search to find the local spanning forest in each hypercube processor, which is later merged to give the complete spanning forest. To the best of our knowledge, WS algorithm is the fastest existing parallel algorithm.

We studied WS algorithm and discovered ways to improve the existing algorithm to give superior results in terms of speedup for sparse graphs. In the next section we will describe our algorithm. The performance of both of these algorithms is presented in the experimental results section.

3.1 An Efficient Hypercube Algorithm for SFP

In this section we first present WS algorithm and then our algorithm with comparison to theirs. We implemented both of these algorithms on an nCUBE II...
computer with 128 processors. Our algorithm exhibits a much better speedup for any graph ranging from partially sparse to dense graphs.

WS algorithm assumes a dense graph and an adjacency matrix representation. Each hypercube processor begins with a partition of the adjacency matrix of $G$. There are two main stages in the algorithm. During the first stage, it computes a local spanning forest using an algorithm shown in Figure 6 under the assumption that the graph has only those edges that are in its partition. Note that the local spanning forest has at most $n - 1$ edges. Once the spanning forest is computed at each processor of the hypercube, the results are sent to other hypercube processors using the hypercube collapsing technique as discussed in Section 1.2.2. A hypercube processor will no longer be active once it has sent the result to another processor.

In the second stage of the algorithm, using the hypercube collapsing technique, the edges of the spanning forest computed during stage one are sent from one half of the hypercube processors to the other half of the hypercube processors. In particular, let $p_1$ and $p_2$ be two processors which have spanning forests $SF_1$ and $SF_2$, respectively, and $p_2$ communicates with $p_1$ during hypercube collapsing. Processor $p_2$ sends $n - 1$ edges of spanning forest $SF_2$ to processor $p_1$. Then, the new $n - 1$ edges are merged using various techniques such as Union and Find and equivalence class algorithms [1, 15] with the $n - 1$ edges of the original spanning forest $SF_1$ generated by $p_1$ during the first stage. This process is continued until we obtain the final result (i.e., the spanning forest $SF$) in processor 0 of the hypercube. Hence, the spanning forest $SF$ is computed in $\log p$ steps, where $p$ is the number of processors.

WS algorithm exhibits good performance with respect to speedup for dense graphs. However, if we have sparse graphs, then the WS algorithm uses more space at every processor of the hypercube. Recall that adjacency matrix of $G$ gets divided into $p$ partitions and thus $\Theta(n^2/p)$ space is needed at every processor.
Their algorithm does not clearly state whether they are reusing the spanning forest $SF_1$ in $p_1$ (explained in the previous paragraph) during merging process of $SF_1$ and $SF_2$. In our algorithm we simply represent $G$ as an adjacency list and the edges are represented by a list $\{(u_i, v_i) \mid (u_i, v_i) \in E\}$. Instead of partitioning the adjacency matrix, we divide the list of edges equally into partitions and send a partition to a processor. This requires $\Theta(m/p)$ space at every processor. Thus for dense graphs (for, graphs having $O(n^2)$ edges), our algorithm uses same space as the one in WS algorithm. But, for sparse graphs our algorithm uses less space. Note that in existing massively parallel machines, such as nCUBE and iPSc's, every processor has only small amount of memory.

In our algorithm we use Union and Find with path compression technique [1] (we refer to this technique as the UFTP) to maintain spanning forest during the various steps. This helps us to retain the original spanning forest generated at each step by adding the new edges to the existing spanning forest rather than generating spanning forest from scratch during every step. We pick a new edge which was received from other processor and check to see whether it forms a cycle in the existing spanning forest using UFTP. The new edge is added to the spanning forest if it does not create a cycle in the spanning forest else it is discarded.

We present an outline of our algorithm for SFP in Figure 7. Initially the edges in the graph $G$ are equally distributed among the hypercube processors, i.e., each processor receives at most $\lceil m/p \rceil$ edges. We notice from Figure 7 that two methods are used to generate the spanning forest, one shown in Step 1 and the other in Step 3. The details for Step 3 are shown in Figure 8.

Similar to WS algorithm, each processor of the hypercube computes a spanning forest using DFS spanning forest algorithm in Step 1 assuming that the only edges that exist in $G$ are the ones in its partition. The edges of the resulting spanning forest at each processor are sent to other processor using hypercube
Algorithm Parallel_Spanning_Forest

Input: Graph $G(V, E)$

Output: Spanning forest $SF$

Step 1: Hypercube processor $PE_j$ receives a list of edges $E_j = \{(u_i, v_i) \mid (u_i, v_i) \in E\}$

($\star E_j \subseteq E; \mid E_j \mid = \lceil \frac{n}{p} \rceil \star$)

Step 2: Processor $PE_j$ computes a local spanning forest based on the information in its list $E_j$ using DFS spanning forest algorithm.

for $i \leftarrow 1$ to $\log p$ do

Step 3: ($\star$ use hypercube collapsing technique to send or receive edges $\star$)

(a) if the $i^{th}$ bit of $PE_j$ is 0 then receive edges from its buddy in the $i^{th}$ dimension;

(b) if the $i^{th}$ bit of $PE_j$ is 1 then send edges to its buddy in the $i^{th}$ dimension;

Step 4: if the $i^{th}$ bit of $PE_j$ is 0 then merge the local spanning forest using the Union and Find technique with the new edges received from other hypercube processor.

endfor

end Parallel_Spanning_Forest

($\star$ Spanning Forest $SF$ is obtained at processor 0 of the hypercube $\star$)

Figure 7. Parallel Spanning Forest Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm *Spanning Forest using Union and Find*

**Input:** Graph $G = (V, E)$

**Output:** Spanning forest $SF = (V, T)$

for each new edge $(u, v)$ received do

$s_u \leftarrow \text{Find}(u)$;

(* $s_u$ is the set name of the set $S_u$ containing $u$ *)

$s_v \leftarrow \text{Find}(v)$;

(* $s_v$ is the set name of the set $S_v$ containing $v$ *)

if $s_u \neq s_v$ then

$\text{Union}(u, v)$;

add edge $(u, v)$ to the spanning forest edges;

endif;
endfor;

end *Spanning Forest using Union and Find*;

Figure 8. Spanning Forest Algorithm Using Union and Find.
collapsing technique. We then use the UFTP log\(p\) times, to find the spanning forest as shown in Step 3 of the main algorithm.

For each processor, in the Union and Find algorithm, the "for" loop is executed for all the edges that are received. Let \((u, v)\) be an edge received. We have \(s_u\) and \(s_v\) to be the set names the sets containing vertices \(u\) and \(v\), respectively. Once we find \(s_u\) and \(s_v\), we check to see whether they are same, i.e., vertices \(u\) and \(v\) belong to the same set. Edge \((u, v)\) will be added to the spanning forest if \(s_u \neq s_v\) and the two sets corresponding to \(s_u\) and \(s_v\) are merged together using Union. An edge is not added to the spanning forest if it belongs to the same set, since adding that edge will create a cycle.

We keep adding the new edges to the existing spanning forest as long as there are no cycles. This helps us eliminate the time taken to merge the two results and not to forget, we do not have to start building the spanning forest from the scratch. Hence, we use the depth first search strategy to build the spanning forest once and we keep adding the new edges using the Union and Find strategy later.

3.2 Theoretical Analysis

As mentioned earlier in the Section 1.1.1, speedup of a \(p\) processor algorithm is the ratio of the time taken by the fastest uniprocessor algorithm \(t_0\) to the time \(t_p\) taken a parallel algorithm using \(p\) processors. In order to theoretically compute the speedup achieved by our algorithm, we will first compute the time taken by the uniprocessor algorithm, in our case it is the time taken by the DFS spanning forest algorithm. Later, we compute the time taken for the parallel algorithm and find the ratio between them.

In order to effectively compute the total time taken by a parallel algorithm, we need to compute the communication time in addition to the computation time. Since communication is a problem by itself, we first consider the computation time
in order to calculate the speedup. We then analyze the communication time, which will give the total time taken by the parallel algorithm in conjunction with the computational time.

Let $G = (V, E)$ be the given graph with $n$ vertices and $m$ edges, the best timing known for any uniprocessor algorithm using depth first search to find spanning forest is $O(n + m)$. In the case of a $p$ processor parallel algorithm, we go through $\log p$ steps in order to compute the spanning forest. We next compute the time taken at each such step.

The first step our algorithm uses the depth first strategy to find the spanning forest. At this step, all the processors compute the spanning forest in parallel, but each processor has at most $\lceil \frac{m}{p} \rceil$ edges instead of $m$ edges as in the case of a uniprocessor algorithm. Hence the time taken in the first step is $O(n + \frac{m}{p})$. During the second step, only half the processors are active and we are also using the Union and Find strategy to find the spanning forest from here on.

The spanning forest computed by all the processors during step one will not have more than $n - 1$ edges. Hence, the other processors will not receive more than $n - 1$ edges. For each edge received, we perform two Find operations and possibly a Union operation. The time taken for a single Union or a single Find is $O(\log n)$, However, the amortized time for a sequence of $O(n)$ Union and Finds is $O(n \alpha(n))$ (details in Section 1.2.1).

It thus takes $O((n - 1)\alpha(n))$ to add the new incoming edges to the existing spanning forest using the Union and Find algorithm. Hence, the time taken at the second step is $O(n \alpha(n))$, the same process continues for all the remaining $\log p - 1$ steps.

Hence, the summation of all the timings is the total computation time taken by the parallel algorithm, which is $O(n + n \alpha(n) \log p + \frac{m}{p})$.

We have the following results from the above, $t_0 = (n + m)$
The speedup of our algorithm varies depending on the character of the input graphs. We have two conditions under which we can study the above speedup relation with respect to the number of vertices \( n \) and the number of edges \( m \) in the given graph. When \( m = n \) (for sparse graphs), we have the speedup to be

\[
S_p = \frac{\frac{2m}{(p-n(a(n))\log p+n)}}{\frac{2p}{(p-n(a(n))\log p+1)}}
\]

\[
= \frac{2p}{(p-n(a(n))\log p+1)}
\]

From the above relation we notice that the speedup decreases for processors greater than 2. Thus, as we increase the number of processors for small size problems, we notice that the efficiency of our algorithm goes down. This phenomena is called the process starvation [10]. Hence, we need to increase the problem size in order to maintain efficiency of our algorithm as we increase the number of processors.

When \( m \gg n \) (for dense graphs), we have the speedup to be

\[
S_p = \frac{\frac{p^m}{(p-n(a(n))\log p+m)}}{\frac{p}{(p-n(a(n))\log p+1)}}
\]

(Since \( n \) is negligible when compared to \( m \))

\[
= \frac{p}{(p-n(a(n))\log p+1)}
\]

We notice from the above relation that for \( m \gg p \), the denominator becomes one and hence the speedup increases proportional to the number of processors. Thus we notice in our algorithm, the speedup is directly dependent on the number of edges \( m \) in the given graph. The speedup increases significantly for problems with very large number of edges in comparison with the number of vertices and the number of processors. Similarly, the speedup increases by fixing the number of processors and increasing the number of edges. We can see similar results from Section 3.3.
Now, let us consider the communication time in our algorithm. Initially, we assume that the edges are equally distributed to all the processors in the hypercube. After Step 1 of our algorithm, half the hypercube processors send \(n - 1\) edges to their buddy processors in parallel. If \(\tau_e\) is the time taken to send one edge from one processor to another processor, it would take \(O(n \tau_e)\) time to send all the \(n - 1\) edges. Since it takes \(\log p\) steps to compute the spanning forest, the total time spent for communication would be \(n \tau_e \log p\). If \(\tau_p\) is the computation time for a basic operation in a hypercube processor, then the total computation and communication time of our algorithm is \(O(n \tau_p \alpha(n) \log p + \tau_p \frac{m}{p} + n \tau_c \log p)\).

Hence, the speedup is,

\[
S_p = \frac{p(n+m)}{(p \cdot \tau_p \alpha(n) \log p + \tau_p \frac{m}{p} + n \tau_c \log p)}
\]

In practice, we always send a set of edges (in the form of packets) from one processor to another in a single communication time unit. However, the maximum number of edges which could be send at a single time depends on the input and output buffer size at each processor. If \(b_s\) is the buffer size, then the time taken to send \(n\) edges is \(O(\frac{n \tau_e}{b_s})\). For the purpose of clarity and simplicity, we chose the value of \(b_s\) as 1 in the above calculations.

### 3.3 Experimental Results

We implemented our algorithm and WS algorithm on an nCUBE-2 parallel computer with 128 processor. We experimentally computed the speedup for these two algorithms for different size of graphs. The results are summarized in Tables 1, 3, 2 and 4. While, comparing both these results, our results are very encouraging and show a better speedup.

In order to test these two algorithms, we generated random graphs by varying the edges and vertices. Both the algorithms were tested for the same set of input data and under similar conditions by fixing the value of \(b_s = 700\). For the
### Table 1

SF: $S_p$ With $V=250$ Using Our Algorithm

<table>
<thead>
<tr>
<th>Edges</th>
<th>number of processors ($p$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0.39</td>
</tr>
<tr>
<td>100</td>
<td>0.76</td>
</tr>
<tr>
<td>500</td>
<td>1.41</td>
</tr>
<tr>
<td>1000</td>
<td>1.63</td>
</tr>
<tr>
<td>3000</td>
<td>1.84</td>
</tr>
<tr>
<td>10000</td>
<td>1.95</td>
</tr>
<tr>
<td>19879</td>
<td>1.97</td>
</tr>
</tbody>
</table>

### Table 2

SF: $S_p$ With $V=250$ Using WS Algorithm

<table>
<thead>
<tr>
<th>Edges</th>
<th>number of processors ($p$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0.14</td>
</tr>
<tr>
<td>100</td>
<td>0.41</td>
</tr>
<tr>
<td>500</td>
<td>0.70</td>
</tr>
<tr>
<td>1000</td>
<td>0.85</td>
</tr>
<tr>
<td>3000</td>
<td>1.28</td>
</tr>
<tr>
<td>10000</td>
<td>1.71</td>
</tr>
<tr>
<td>19879</td>
<td>2.22</td>
</tr>
<tr>
<td>Edges</td>
<td>number of processors (p)</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>0.52</td>
</tr>
<tr>
<td>990</td>
<td>1.30</td>
</tr>
<tr>
<td>4950</td>
<td>1.76</td>
</tr>
<tr>
<td>14844</td>
<td>1.19</td>
</tr>
<tr>
<td>39596</td>
<td>1.96</td>
</tr>
<tr>
<td>98984</td>
<td>1.99</td>
</tr>
<tr>
<td>198926</td>
<td>1.99</td>
</tr>
<tr>
<td>242180</td>
<td>1.99</td>
</tr>
</tbody>
</table>

Table 4
SF: $S_p$ With V=700 Using WS Algorithm

<table>
<thead>
<tr>
<th>Edges</th>
<th>number of processors (p)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>0.50</td>
</tr>
<tr>
<td>990</td>
<td>0.66</td>
</tr>
<tr>
<td>4950</td>
<td>1.51</td>
</tr>
<tr>
<td>14844</td>
<td>1.68</td>
</tr>
<tr>
<td>39596</td>
<td>1.89</td>
</tr>
<tr>
<td>98984</td>
<td>2.18</td>
</tr>
<tr>
<td>198926</td>
<td>2.49</td>
</tr>
<tr>
<td>242180</td>
<td>2.58</td>
</tr>
</tbody>
</table>
comparison sake and for simplicity we consider only the computational timings to compute the speedup for both the algorithms. Observe that communication time is same in both the algorithms. Just to give a flavor of our results we have presented the results for 250 and 700 vertices even though both the algorithms were tested for various graphs. The resulting speedup for other graphs are shown in Figure 9 to Figure 18.

Before going into the detail of comparing both these algorithm, let us first look at some anomalies about the parallel computing using the give table. When we observe the speedup using 500 edges and 250 vertices in Table 1, we notice that the speedup increases form 1.41 (2 processors) to 2.71 (16 processors) and later starting to decline. This is due to processor starvation. While, the speedup gradually increases from 1.97 (2 processors) to 29.40 (128 processors) in the case of 19879 edges from the same table. Hence, from these results it is very clear that, we have to increase the problem size as we increase the number of processors [10], in order to fully utilize the processing power of 128 processors. We also notice the similar results by looking at the speedup data for smaller size problems, in fact we do not have any speed up for 10 and 100 edges in Table 1. Hence, it can also be stated that the efficiency of the algorithm is poor for graphs with larger number of vertices and less number of edges.

We notice from the given tables, that both the algorithms were tested for sparse to dense graphs for different vertices. In general, our algorithm shows better speedup for sparse graphs while compared with WS algorithm. If we compare Table 1 and Table 2, we notice that their algorithm does not give any speedup till it reaches 10000 edges, while our algorithm starts to give a reasonable amount of speedup for sparse graphs starting from 500 edges and 250 vertices.

When we look at Table 3 and Table 4, we notice our algorithm consistently gives a good speedup from 990 edges. It starts with a speed up of 1.30 using 2 processors and consistently increases to 2.35 using 32 processors and starts to fall
down again for 64 and 128 processor. This is again due to processor starvation. If we look at Table 4, we notice processor starvation until 39596 edges after that stage, we notice their algorithm giving a much better speedup for less number of processors (less than 16 processors). This is mainly due to the reason that WS algorithm was designed for dense graphs. Thus we see that WS algorithm works satisfactorily for dense graphs while ours work good for both the sparse and dense graphs. In general we notice an important fact from all these tables that both algorithm work better for large size problem, as we increase the number of edges we reduce processor starvation, which immediately improves the speedup.
Figure 9. SF:$S_p$ for $V=250$ Using Our Algorithm.

Figure 10. SF:$S_p$ for $V=250$ Using WS Algorithm.
Figure 11. SF:S_p for V=700 Using Our Algorithm.

Figure 12. SF:S_p for V=700 Using WS Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 13. SF:$S_p$ for $V=125$ Using Our Algorithm.

Figure 14. SF:$S_p$ for $V=125$ Using WS Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 15. SF:S for V=350 Using Our Algorithm.

Figure 16. SF:S for V=350 Using WS Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 17. SF: $S_p$ for $V=600$ Using Our Algorithm.

Figure 18. SF: $S_p$ for $V=600$ Using WS Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER IV

MINIMUM COST SPANNING FOREST

Let $G = (V,E)$ be an undirected weighted graph with costs on the edges. The problem of constructing a spanning forest $SF$ with the least possible total cost is viewed as the minimum cost spanning forest problem, we refer to this problem as MCSFP and the spanning forest generated as MCSF. The total cost of the spanning forest is the sum of the cost of its edges. This problem has various practical applications, such as in VLSI for routing problems, in networks for efficient broadcasting, in connecting different cities by highways with the minimum distance and in many other problems [16, 23, 25, 31].

There exist several sequential algorithms to solve MCSFP in $O((m + n) \log n)$ time with a space requirement of $O(n + m)$, where $m$ and $n$ are the number of edges and vertices in the given graph, respectively. For problems with sufficiently large value of $m$, we can find the solutions in $O(m \log n)$ time. Some of the standard methods to solve this problem using sequential models of computation are Kruskal's method [16], Prim's method [24], Dijkstra's method [9] and Sollin's method [31].

Let us go over one of the sequential algorithm namely the Kruskal's algorithm before we present the parallel algorithms. An outline of the Kruskal's algorithm using Union and Find is given in Figure 19. Kruskal's algorithm assumes an adjacency list representation of $G$ as input. In the figure, $NE$ is a counter in the algorithm which keeps track of the number of edges added to the minimum cost spanning forest. Initially the edges $(u,v) \in E$ are sorted in ascending order and placed in the priority queue $Q$ and each vertex is initialized to be in its own set.
At each stage of the while loop, the least cost edge \((u, v)\) is removed from \(Q\). The next step is to find the set names \(s_u\) and \(s_v\) where \(u\) and \(v\) belong respectively. The edge \((u, v)\) is added to the spanning forest if \(S_u\) is different from \(S_v\) and the corresponding sets \(s_u\) and \(s_v\) are merged. If both the vertices are in the same set, then it implies that adding edge \((u, v)\) creates a cycle in the spanning forest. Thus, the edge is discarded. The while loop continues until all the edges in \(Q\) are used or until we get the complete minimum cost spanning tree. Kruskal's algorithm yields a minimum cost spanning tree if there exists a defined path between all the vertices in \(G\) otherwise it yields a minimum cost spanning forest.

Parallel solutions for finding a minimum cost spanning forest has been extensively studied \([2, 4, 14, 18, 19, 22, 23, 34]\). An \(O(\log^2 n)\) time algorithm on an \(n \times n\) mesh of trees based parallel computer has been presented by Leighton \([19]\), where \(n\) is the number of vertices in the given graph \(G\). Leighton's algorithm is very similar to the well known sequential algorithm to compute minimum cost spanning forest using Sollin's technique \([31]\). Initially the algorithm assumes every vertex to be a supervertex. In general a supervertex \(SV = \{u_1, u_2, \ldots, u_k\}\) such that \(u_i \in V\) and \(u_i\) and \(u_j\) is connected by a path in \(G\) for \(i \neq j\). A supervertex \(SV\) is identified by a leader, which is, in general, either the lowest or the highest numbered vertex belonging to the set \(SV\).

The basic idea of the Leighton's algorithm is to collect supervertex and merge them using the minimum cost edges between them, which creates larger supervertex. This is called the coalescing operation. During this operation, there may be long chains of vertices in each supervertex, which may compete with one another and create cycle in MCSF. This could be avoided by pointing each vertex to its leader and the leader pointing to itself in each supervertex. Thus, two vertex with the same leader will not be merged as one supervertex. The long chain of vertices in each supervertex is condensed to point to its leader so that its leader
Algorithm *Minimum_Cost_Spanning_Forest*

**Input:** An undirected weighted graph $G = (V, E)$

**Output:** Minimum Cost Spanning Forest $MCSF$

Construct a priority queue $Q$ containing all edges in $E$ such that they are in non-decreasing order of their costs;

Let $S_i = \{v_i\}$, for $1 \leq i \leq n$;

$NE \leftarrow 0$; (* Initializing number of edges in MCSF to 0 *)

**while** $(Q \neq \text{empty})$ and $(NE \leq n - 1)$ **do**

Choose edge $(u, v)$ in $Q$ with the lowest cost;

Delete edge $(u, v)$ from $Q$;

$s_u \leftarrow \text{Find}(u)$;

(* $s_u$ is the set name of the set $S_u$ containing $u$ *)

$s_v \leftarrow \text{Find}(v)$;

(* $s_v$ is the set name of the set $S_v$ containing $v$ *)

**if** $s_u \neq s_v$ **then**

$\text{Union}(S_u, S_v)$;

add edge $(u, v)$ to the spanning forest $MCSF$;

$NE \leftarrow NE + 1$;

endif;

endwhile;

end *Minimum_Cost_Spanning_Forest*;

Figure 19. Kruskal’s Minimum Cost Spanning Forest Algorithm.
could be found in one step. The coalescing operation continues until there is no change in any supervertex.

In the next section, we will present our algorithm, which is essentially a parallelization to Kruskal's minimum cost spanning forest algorithm. Our algorithm uses \( p \) hypercube processors and takes \( \log p \) steps to compute the minimum cost spanning forest. The speedups produced by our algorithm are very encouraging. An important extension of this algorithm could be in the area of shortest path problems. Several parallel algorithms have been presented to find shortest path in a weighted directed graph \( G \) \([7, 26]\). Most of them use matrix multiplication and use \( n^3 \) processors to find an efficient solution. For example, in order to find a solution for a graph having 75 vertices, we will need at least 421875 processors to get an efficient solution, which is not very practical. Since our algorithm uses \( p \) processors, for any \( p \) independent of \( n \), one of our future work would be to adapt this algorithm to the shortest path problems.

4.1 An Algorithm for MCSFP

In this section, we present a parallel algorithm to solve MCSFP. Our algorithm is a parallel version of Kruskal's algorithm. We essentially use the algorithm shown in Figure 19 and we show how this algorithm is implemented on a hypercube parallel computer with \( p \) processors.

An outline of our algorithm is shown in Figure 20. During the preprocessing step of our algorithm, we sort the edges \((u_i, v_i) \in E\), in ascending order by their associated costs \( c_i \)'s once. Then the edge list \( \{(u_i, v_i, c_i) \mid (u_i, v_i) \in E \& c_i = cost(u_i, v_i)\} \) is equally partitioned over all the processors as used in Step 1 of our algorithm. Thus each processor receives \( O(m/p) \) edges and hence the space required at each processor is \( O(m/p) \).
Algorithm *Parallel_Minimum_Cost_Spanning_Forest*

**Input:** Graph $G = (V, E)$

**Output:** Minimum Cost Spanning forest $MCSF$

**Step 1:** Hypercube processor $PE_j$ receives a list of edges $E_j = \{(u_i, v_i, c_i) \mid (u_i, v_i) \in E\}$

(* $E_j \subseteq E; \mid E_j \mid = \lceil \frac{m}{p} \rceil \star$*)

The edges in $E_j$ are sorted according to the cost $c_i$'s.

**Step 2:** Processor $PE_j$ computes a local minimum cost spanning forest based on the information in its list $E_j$ using Kruskal's algorithm of Figure 19.

for $i = 1 \to \log p$ do

**Step 3:** (* use hypercube collapsing technique to send or receive edges *)

(a) if the $i^{th}$ bit of $PE_j$ is 0 then receive edges from its buddy in the $i^{th}$ dimension;

(b) if the $i^{th}$ bit of $PE_j$ is 1 then send edges to its buddy in the $i^{th}$ dimension;

**Step 4:** if the $i^{th}$ bit of $PE_j$ is 0 then merge the local minimum spanning forest using the Union and Find technique with the new edges received from other hypercube processor.

**Step 5:** if $PE_j = PE_0$ and number of edges in the local minimum cost spanning forest $= n - 1$

then stop;

endfor

end *Parallel_Minimum_Cost_Spanning_Forest*

Figure 20. Parallel Minimum Cost Spanning Forest Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
When we distribute the list of edges to each processor, we make sure that the first list with the smallest \( \left\lfloor \frac{n}{p} \right\rfloor \) edge costs goes to processor 0 and the next list goes to processor 1 and so on for all the processors in the hypercube. Hence, processor number \( p - 1 \) will have the list of edges with the highest \( \left\lfloor \frac{n}{p} \right\rfloor \) costs. The main reason for doing this is due to the fact that we will be receiving the complete MCSF in processor 0. This will make sure that the resulting MCSF will have the least total cost.

In Step 2 of the algorithm, each processor computes a local minimum cost spanning forest using Kruskal’s algorithm assuming that the graph has only those edges in its edges list. The edges in the newly formed minimum cost spanning forest are sent from one half of the hypercube processors to the other half of the hypercube processor using hypercube collapsing technique. Note here that if processor \( p_i \) sends edges to processor \( p_j \) of the hypercube during hypercube collapsing, it is always made sure that \( i > j \). Thus, the edge cost of the local minimum cost spanning forest in processor \( p_j \) will be lesser than the costs of the incoming edges.

Once we have the local minimum cost spanning forest in each processor, we merge these spanning forest in \( \log p \) steps to form MCSF in processor 0 of the hypercube. We use Union and Find with path compression to merge the incoming edges with the existing local spanning forest at each processor. The details of the Union and Find algorithm were shown in Figure 8 and the steps were explained in Section 3.1.

4.2 Theoretical Analysis

The speedup \( S_p \), which measures the quality of a \( p \) processor algorithm is the ratio of the time taken by a fastest sequential algorithm \( t_0 \) to the time taken
by parallel algorithm $t_p$ using $p$ processors. In the case of a standard sequential algorithm, the computational time will be the total computation time spent by the algorithm unlike a parallel algorithm where the total time is the sum of communication time between processors and the max computation time at any processor. The best known sequential time $t_v$ for a uniprocessor algorithm to solve MCSF with $m$ edges is $O((m + n) \log n)$.

We present the total time taken by our parallel algorithm at the end of this section. Each processor in the hypercube, assumes that the list of edges it receives are sorted in ascending order by its edge costs. Hence, our algorithm has to perform preprocessing in order to sort these edges before they are distributed to all the hypercube processors. It takes $O(m \log n)$ to sort $m$ edges in $G$ during the preprocessing step.

During Step 2 of our algorithm, each hypercube processor receives $\left\lceil \frac{m}{p} \right\rceil$ sorted edges. Since, Union and Find technique is used to find the local minimum cost spanning forest for $\left\lceil \frac{m}{p} \right\rceil$ edges, it takes $O(\frac{m}{p} \alpha(\frac{m}{p}))$ time (details of this result is explained in Section 3.2) for Step 2. The local minimum cost spanning forest does not have more than $n - 1$ edges, which are sent to other hypercube processor using hypercube collapsing technique. In Step 4, we merge the local minimum cost spanning forest using Union and Find with path compression with the incoming $n - 1$ edges, which takes $O(n \alpha(n))$ time. This process is repeated $\log p$ times. Hence, we have the total computation time to be $O(m \log n + \frac{m}{p} \alpha(\frac{m}{p}) + n \alpha(n) \log p)$.

From the above discussion we have the following,

\[
t_v = (m + n) \log n,
\]
\[
t_p = (m \log n + \frac{m}{p} \alpha(\frac{m}{p}) + n \alpha(n) \log p),
\]
\[
S_p = \frac{t_0}{t_p} = \frac{((m+n) \frac{\log n}{p})}{(m \log n + \frac{m}{p} \alpha(\frac{m}{p}) + n \alpha(n) \log p)}.
\]

Usually, finding MCSF is part of a bigger problem and hence we may disregard the preprocessing time for the computation of speedup. Hence, the speedup without preprocessing is,
Let us asymptotically analyze the behavior of the speedup relationship by varying the number of edges $m$ and the number of vertices $n$. When $n = m$ we have the speedup relation as

\[
S_p \approx \frac{(2m \log m)}{(m \alpha(\frac{m}{p}) + p \alpha(m) \log p)}
\]

Thus from the above relation, we notice that the speedup decreases by fixing the number of vertices and increasing the number of processors. However, we notice that when $m >> p$, we still get good speedup. Hence, irrespective of whether the graph is sparse or dense, the speedup increases significantly for large number of edges.

When $m >> n$, we have the speedup to be

\[
S_p \approx \frac{(m \log m)}{(m \alpha(\frac{m}{p}) + p \alpha(m) \log p)} \quad \text{(Since $m$ is significantly large compared to $n$)}
\]

We notice from the above relation that, for $m >> p$ the speedup increases as we increase the number of processors. We also notice that the speedup increases by fixing the number of processors and increasing the number of edges. In general the algorithm directly depends on the number of edges $m$ in the given graph. We notice, similar behavior in our algorithm by observing the tables presented in the experimental section.

In order to effectively compute the total time of the parallel algorithm, we have to consider the communication time along with the computation time. We assume that the edges are already available in each processor after preprocessing step. After we find the minimum cost spanning forest in Step 2, we have at most $n - 1$ edges in each processor. If $\tau_c$ is the communication time to send one edge,
it would take $O(n \tau_c)$ to send all $n - 1$ edges. In general a "chunk" of edges are communicated at a time from one processor to another. The number of edges that can be communicated in one time unit depends on the input and output buffer sizes at each processor. If the buffer size is $b_s$ then the time taken to send $n$ edges is $O(\frac{n}{b_s})$. For the sake of simplicity, we assume the buffer size $b_s = 1$.

We notice from our algorithm that the minimum cost spanning forest is formed during $\log p$ hypercube collapsing steps, the total time spent in communication is thus be $O(n \tau_c \log p)$. Similarly if $\tau_p$ is the computation time of a single basic operation in a hypercube processor, then the total time taken by our algorithm is $O(\frac{m}{p} \tau_p \alpha(\frac{m}{p}) + n \tau_p \alpha(n) \log p + n \tau_c \log p)$.

Hence, the speedup is,

$$S_p = \frac{\frac{(m+n)}{p} \log n}{(m \tau_p \alpha(m) + n \tau_p \alpha(n) \log p + n \tau_c \log p)}$$

### 4.3 Experimental Results

We implemented our algorithm on an nCUBE II parallel computer with 128 processor. The results of our algorithm is summarized in Table 5 and 6. Our results show good speedup which encourages us to use similar idea to solve shortest path problems.

We generated random inputs ranging from sparse to dense graphs by varying the edges. Speedup are calculated based on the computation time. We chose the value of $b_s$ to be 700 after several experiments. Thus we could send 700 edges from one processor to another in $\tau_c$ time unit (discussed in the earlier section). We chose $b_s = 700$ due to the limited size of the input buffer at the lower numbered processors of our hypercube computer. Varying the value of $b_s$ will not affect the computation time, but it will affect the communication time and hence the total time taken by the parallel algorithm. Speedup tables for graphs with 250 and 700
Table 5
MCSF: $S_p$ With $V=250$

<table>
<thead>
<tr>
<th>Edges</th>
<th>number of processors ($p$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0.83</td>
</tr>
<tr>
<td>100</td>
<td>0.95</td>
</tr>
<tr>
<td>500</td>
<td>1.07</td>
</tr>
<tr>
<td>1000</td>
<td>1.23</td>
</tr>
<tr>
<td>3000</td>
<td>1.59</td>
</tr>
<tr>
<td>10000</td>
<td>1.86</td>
</tr>
<tr>
<td>19879</td>
<td>1.93</td>
</tr>
</tbody>
</table>

Table 6
MCSF: $S_p$ With $V=700$

<table>
<thead>
<tr>
<th>Edges</th>
<th>number of processors ($p$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>0.96</td>
</tr>
<tr>
<td>990</td>
<td>0.98</td>
</tr>
<tr>
<td>4950</td>
<td>1.42</td>
</tr>
<tr>
<td>14844</td>
<td>1.73</td>
</tr>
<tr>
<td>39596</td>
<td>1.89</td>
</tr>
<tr>
<td>98984</td>
<td>1.95</td>
</tr>
<tr>
<td>198926</td>
<td>1.98</td>
</tr>
<tr>
<td>242180</td>
<td>1.98</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
vertices are shown in Table 5 and 6 and the speedup for other size graphs are shown in Figure 21 to Figure 25.

In Table 5 we see that the behavior of our algorithm is very close to the sequential algorithm for sparse graphs, by looking at edges 10 to 1000. The reason for this behavior is due to process starvation [10]. On the other hand, the speedup increases towards a greater extent from 3000 edges and reaches a maximum of 9.82 for 19879 edges using 128 processor.

We see similar behavior in our algorithm from Table 6 for small size problem as described earlier. However, we get a good improvedment in the speedup for edges greater than 4950 in Table 6. If we closely observer the speedup from both the tables, we notice one common fact. The algorithm really does not depend on the number of vertices, it only depends on the number of edges in the given graph. Our algorithm gives good speedup in general and for large problem size in particular with respect the number of edges $m$ in $G$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 21. MCSF:S, for V=250.

Figure 22. MCSF:S, for V=700.
Figure 23. MCSF: $S_p$ for $V=125$.

Figure 24. MCSF: $S_p$ for $V=350$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 25. MCSF $S_p$ for $V=600$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER V

CONNECTED COMPONENTS

An undirected graph $G = (V, E)$ is said to be connected if for every pair $v_i$ and $v_j$ of its vertices there is a path from $v_i$ to $v_j$. A connected component $CC$ of a graph $G$ is a subgraph $G'$ of $G$ that is connected. Thus, the problem of finding the connected components in the given graph $G$ is to label all the vertices in the connected subgraph $G'$, with some unique label. We refer to this problem as CCP. Connected component problem CCP has wide applications in fields such as graph theory, pattern recognition etc. There are several sequential algorithms [1] and parallel algorithms [5, 10, 14, 20, 22, 30, 32] exist to find solutions to CCP based on breadth first search, transitive closure, vertex collapse, matrix multiplication etc.

Let us go over a sequential algorithm to solve CCP using depth first search. This algorithm is very similar to the algorithm shown in Figure 6. Since, we are using depth first search technique to solve this problem, we will refer to this algorithm as DFS connected component algorithm. A formal version of DFS connected component algorithm is shown in Figure 26.

The basic idea of the DFS connected component algorithm is to give a unique label to a set of vertices which belong to the same component in the given graph $G$. We start the algorithm by marking all the vertices to "new" and set the label counter to 0. We next pick a vertex $v_i$ which is initially labeled "new" and execute the procedure DFSCC. The parameter for this procedure is the unmarked vertex and the label number.

We start the procedure DFSCC by marking the new vertex $v_i$ to the given label and pick an adjacent vertex $u$ of $v$ which is originally marked "new" and call
Procedure \textit{DFSCC}(v, l)
(*) v a starting vertex and a label l *)

Input: Graph $G = (V, E)$

Output: Connected Components

mark \textit{v} with label \textit{l}

for each vertex \textit{u} adjacent to \textit{v} do

if \textit{u} is marked “new” then DFSCC(u, \textit{l}) endif

endfor

end DFSCC

Algorithm \textit{DFS\_Connected\_Component}

Input: Graph $G = (V, E)$

Output: Connected Components $CC(V)$

mark $v_i$ ← “new”, for $1 \leq i \leq n$;

\textit{label} ← 0;

for $i ← 1$ to $n$ do

if $v_i$ is marked “new” then

\textit{label} ← \textit{label} + 1;

DFSCC($v_i$, \textit{label})

endif

endfor

end DFS\_Connected\_Component

Figure 26. DFS Connected Component Algorithm.
the procedure DFSCC recursively with the same label. This process is continued until all the vertex belonging to that component has been labeled with the same label. Once the first connected component has been found, procedure DFSCC returns to the main algorithm and starts with a new vertex which is still marked “new.” The algorithm continues until there are no vertices marked “new.” DFS connected component algorithm takes $O(n + m)$ time with a space complexity of $O(n + m)$ if we represent $G$ using an adjacency list.

Parallel solutions to this problem have been developed using similar ideas as described in Chapter III and Chapter IV. All the algorithms described in the earlier chapter could be very easily adapted to solve CCP without any change in the time complexity. Leighton [19] has presented an $O(\log^2 n)$ time algorithm on an $n \times n$ mesh of trees. This algorithm is a modified version of his minimum cost spanning forest algorithm. Leighton’s algorithm assumes equal weight for existing edges and infinite weight for nonexisting edges. Initially the algorithm assumes each vertex to be a connected component and addressed as a supervertex. As discussed earlier, each supervertex has a unique label.

Leighton’s algorithm starts to collect supervertices and merges them into large supervertices if it finds an edge connecting two supervertices. This operation is known as the coalescing operation. The coalescing operation is continued until there is no change in the supervertex. Since, each supervertex has a unique label, all the vertices in that supervertex are recognized using its supervertex label.

Woo and Sahni have developed a $p$ processor algorithm to solve CCP. This algorithm is referred as the WS algorithm. Their algorithm is very clearly described in Chapter III and in Section 3.1. We found their algorithm to be the fastest existing parallel algorithm which uses only $p$ processors. Their algorithm works well for dense graphs in general, we modified their algorithm so that it gives good speedup for any graph ranging from sparse to dense. In the next section we will be directly presenting our algorithm since WS algorithm was described in
Section 3.1. The performance of both these algorithms are tested and summarized in the experimental results section.

5.1 An Algorithm for CCP

Our main objective in this algorithm is to efficiently compute the connected components in a given graph with the best possible speedup. This algorithm is very similar to the one described in Section 3.1, thus we can see the performance of both these algorithms to be very close if we compare experimental results.

As used in earlier chapters we use two methods namely the breadth first search technique and Union and Find technique to compute the connected components for a given graph $G$. Our algorithm is formally presented in Figure 27. Initially each hypercube processor receives at most $\left\lceil \frac{m}{p} \right\rceil$ edges and starts building a spanning forest using DFS spanning forest algorithm based on the information in its local memory.

In Step 3 of the main algorithm, we send the list of edges in the spanning forest from one half of the hypercube processors to other half using hypercube collapsing technique. Each hypercube processor which has received the list of edges, starts to merge the new edges with its spanning forest using Union and Find with path compression technique. The details of the Union and Find algorithm is shown in Figure 8 and explained in Section 3.1. After $\log p$ steps, we have the final spanning forest at processor 0 of the hypercube.

However, we are interested in finding the connected components rather that finding the spanning forest. If we notice the Union and Find algorithm from Figure 8, we merge all vertices which belong to the same component to one set. Hence, at the end of the algorithm, the number of components found will be equal to the number of nonempty sets and each vertex will be identified by the set number which would be its label and component number.
**Algorithm** *Parallel.Connected.Component*

**Input:** Graph $G = (V, E)$

**Output:** Connected Components: a label for every vertex

*Step 1:* Hypercube processor $PE_j$ receives a list of edges $E_j = \{(u_i, v_i) | (u_i, v_i) \in E\}$;

(* $E_j \subseteq E; \mid E_j \mid = \lceil \frac{\log p}{\log 2} \rceil \star$*)

*Step 2:* Processor $PE_j$ computes a local spanning forest based on the information in its list $E_j$ using DFS spanning forest algorithm.

*for* $i \leftarrow 1$ *to* $\log p$ *do*

*Step 3:* (*use hypercube collapsing technique to send or receive edges*)

(a) if the $i^{th}$ bit of $PE_j$ is 0 then receive edges from its buddy in the $i^{th}$ dimension;

(b) if the $i^{th}$ bit of $PE_j$ is 1 then send edges to its buddy in the $i^{th}$ dimension;

*Step 4:* if the $i^{th}$ bit of $PE_j$ is 0 then merge the local spanning forest using the Union and Find technique with the new edges received from other hypercube processor.

*endfor*

**end Parallel.Connected.Component**

(* Connected Component is obtained at processor 0 of the hypercube *)

---

Figure 27. Parallel Connected Components Algorithm.
5.2 Theoretical Analysis

We notice from the above algorithm that it is very similar to our algorithm discussed in Section 3.1 with little modification to compute connected components. Hence, the time complexity for this algorithm is same as the one given in Section 3.2.

Thus we have the following from Section 3.2

\[ t_0 = (n + m) \]
\[ t_p = (n \alpha(n) \log p + \frac{m}{p}) \]
\[ S_p = \frac{t_0}{t_p} = \frac{\log p + \frac{m}{p}}{(n + m) \log p + \frac{m}{p}} \]

where \( t_0 \) is the time taken by an uniprocessor algorithm to solve CCP, \( t_p \) is the time taken by a \( p \) processor algorithm without including the communication time and \( S_p \) is the speedup.

The speedup of our connected component algorithm behaves very similar to the spanning forest algorithm described in Chapter III. In Section 3.2, we have asymptotically analyzed the behavior of the speedup by varying the number of edges \( m \) and the number of vertices \( n \). In general, for sparse graphs \((n = m)\), the speedup does not increase significantly as we increase the number of processors. However, we have a good speedup for larger size problems when \( m \gg n \). We observe from the discussion in Section 3.2 that the speedup increases for \( m \gg p \) and when the number of edges are increased for fixed set of processors.

If we consider the communication time in addition to the computation time, we have the total time \( t_p \) from Section 3.2 as following,

\[ t_p = O(n \tau_p \alpha(n) \log p + \frac{m}{p} + n \tau_c \log p) \]

\( \tau_p \) is the computation time for a basic operation in a hypercube processor and \( \tau_c \) is the time taken to send one edge from one processor to another processor. Notice that in a hypercube processor, we can send a list of edges in one communication time unit \( \tau_c \) depending on the local memory size of the input and output.
buffers at each processor. If $b_s$ is the buffer size then the time taken to send $m$ edges is $O(m/b_s)$. We may assume, for simplicity, the buffer size $b_s = 1$ when we calculate the communication time for our algorithm. Hence, the resulting speedup including the communication time is

$$S_p = \frac{p^{(u+m)}}{(p+\tau_p \alpha(n) \log p + m \tau_p + c \log p)}$$

5.3 Experimental Results

We implemented WS algorithm and our algorithm on nCUBE II (with 128 processors) and executed the algorithm for different size graphs by varying the vertices and edges. The results are summarized in Tables 7, 9, 8 and 10 and the speedup for other size graphs are shown in Figure 28 to Figure 37.

If we closely compare the tables presented in this section with the tables given in Section 3.3, we notice that the resulting speedup are very close and hence resulting in a similar behavior. This is mainly because, the structure of both the algorithms is just the same with a little modification in the connected component algorithm to give the label numbers for each vertex. Thus, the discussion presented in Section 3.3 equally holds for our connected component algorithm as well.

The algorithm were tested for various graphs for size ranging from 250 to 700 vertices with a value for $b_s = 700$. In general, our algorithm gives a good speedup for graphs varying from sparse to dense while their algorithm gives good speedup for dense graphs. The maximum speedup using our algorithm for 700 vertex is 86.39 while WS algorithm gives 25.32.
Table 7

CC: $S_p$ With V=250 Using Our Algorithm

<table>
<thead>
<tr>
<th>Edges $e$</th>
<th>number of processors ($p$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0.41</td>
</tr>
<tr>
<td>100</td>
<td>0.79</td>
</tr>
<tr>
<td>500</td>
<td>1.45</td>
</tr>
<tr>
<td>1000</td>
<td>1.65</td>
</tr>
<tr>
<td>3000</td>
<td>1.86</td>
</tr>
<tr>
<td>10000</td>
<td>1.97</td>
</tr>
<tr>
<td>19879</td>
<td>1.99</td>
</tr>
</tbody>
</table>

Table 8

CC: $S_p$ With V=250 Using WS Algorithm

<table>
<thead>
<tr>
<th>Edges $e$</th>
<th>number of processors ($p$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0.16</td>
</tr>
<tr>
<td>100</td>
<td>0.43</td>
</tr>
<tr>
<td>500</td>
<td>0.72</td>
</tr>
<tr>
<td>1000</td>
<td>0.89</td>
</tr>
<tr>
<td>3000</td>
<td>1.32</td>
</tr>
<tr>
<td>10000</td>
<td>1.75</td>
</tr>
<tr>
<td>19879</td>
<td>2.32</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Table 9

CC: $S_p$ With $V=700$ Using Our Algorithm

<table>
<thead>
<tr>
<th>Edges</th>
<th>number of processors $(p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>0.53</td>
</tr>
<tr>
<td>990</td>
<td>1.32</td>
</tr>
<tr>
<td>4950</td>
<td>1.79</td>
</tr>
<tr>
<td>14844</td>
<td>1.21</td>
</tr>
<tr>
<td>39596</td>
<td>1.99</td>
</tr>
<tr>
<td>98984</td>
<td>2.01</td>
</tr>
<tr>
<td>198926</td>
<td>2.01</td>
</tr>
<tr>
<td>242180</td>
<td>2.03</td>
</tr>
</tbody>
</table>

Table 10

CC: $S_p$ With $V=700$ Using WS Algorithm

<table>
<thead>
<tr>
<th>Edges</th>
<th>number of processors $(p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>0.53</td>
</tr>
<tr>
<td>990</td>
<td>0.69</td>
</tr>
<tr>
<td>4950</td>
<td>1.57</td>
</tr>
<tr>
<td>14844</td>
<td>1.73</td>
</tr>
<tr>
<td>39596</td>
<td>1.92</td>
</tr>
<tr>
<td>98984</td>
<td>2.21</td>
</tr>
<tr>
<td>198926</td>
<td>2.52</td>
</tr>
<tr>
<td>242180</td>
<td>2.62</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 28. CC: $S_p$ for $V=250$ Using Our Algorithm.

Figure 29. CC: $S_p$ for $V=250$ Using WS Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 30. CC:$S_p$ for $V=700$ Using Our Algorithm.

Figure 31. CC:$S_p$ for $V=700$ Using WS Algorithm.
Figure 32. CC: $S_p$ for $V=125$ Using WS Algorithm.

Figure 33. CC: $S_p$ for $V=125$ Using Our Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 34. CC: $S_p$ for $V=350$ Using WS Algorithm.

Figure 35. CC: $S_p$ for $V=350$ Using Our Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 36. CC:$S_p$ for $V=600$ Using Our Algorithm.

Figure 37. CC:$S_p$ for $V=600$ Using WS Algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER VI

CONCLUSION AND FUTURE RESEARCH

In this thesis, we have studied various graph problems such as spanning forest problem, minimum cost spanning forest problem and connected components problem. We have seen the conventional way of finding solutions to these problems by using various sequential algorithms. We have also presented different existing parallel algorithms used to solve these problems. For each of these problems described above, we have presented our parallel algorithms for the hypercube parallel computer using \( p \) processors. Our algorithm does not demand large number of processors. All of our algorithms are efficient in terms of space requirement and speedup \( S_p \). We have theoretically analyzed the behavior of our algorithms and presented the experimental results for each algorithm.

Many of the graph related problems such as transitive closure, shortest path problem, bi-connected components, etc, can be solved using similar methods we have seen in this thesis. Since our solutions for the three graph problems produce superior results in term of speedup, we could modify the existing algorithm for other graph problems as well. Furthermore, many efficient solutions for the shortest path problems uses large number of processors, which are not very practical. However, the solution we presented to solve minimum cost spanning tree could be adapted to solve the shortest path problems. Hence, we can present an efficient solution irrespective of the number of processors for the shortest path problems.

Our next research area would be to efficiently utilize the processing power. If we carefully go over the hypercube collapsing technique, we notice that it takes \( \log p \) steps to reduce a \( p \)-processor hypercube to 0 dimension. In all our algorithms,
we spend \( \log p \) such steps to obtain the final solution in node 0 of the hypercube. We also emphasize the fact that once an hypercube node participates in sending the results to other hypercube node, it remains dormant until the final result is obtained in node 0 of the hypercube. Thus, at each hypercube collapsing stage we loose the processing power of half the hypercube nodes. This would stimulate our research interest to find solutions to this problem so that we can efficiently use the released processors in the rest of the computation process.
REFERENCES


76


