Order Number 9126227

Divide-and-conquer algorithms for multiprocessors

Mukkavilli, Lakshmankumar, Ph.D.

Iowa State University, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Divide-and-conquer algorithms for multiprocessors

by

Lakshmankumar Mukkavilli

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1991

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I am grateful to my advisor Dr. Gurpur M. Prabhu for his advice and help. I would like to thank the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory for providing me access to their computing facilities.

It is not a tradition in India to thank one's wife in public because the wife is considered a part of the husband (*ardhangi*[1]) and any endeavour the husband undertakes, the wife is deemed to be a participant in it. Lakshmi has been a great partner.

[1] In Sanskrit *ardh* means half and *ang* means body.

CHAPTER 1. INTRODUCTION

Many problems in science and engineering are becoming so computationally demanding that conventional sequential computers can no longer provide the required computing power. Researchers and application programmers are turning to machines with parallel computing capability which have the potential to provide that power. During the past decade, there has been a tremendous surge in understanding the nature of parallel computation [50,56,67,27]. The field of parallel computing has opened up new possibilities in the areas of algorithms, language and language extensions, and architectures [31,10,42,5]. A large number of parallel computers are commercially available. Shared memory parallel computers include Multiple Instruction Multiple Data (MIMD) machines such as Alliant, Cray, Encore, and Sequent. Distributed memory computers include MIMD machines such as Intel hypercube, Warp, and Transputer, and Single Instruction Multiple Data (SIMD) machines such as Connection machine, Distributed Array Processor, and Maspar. Use of these computers has been demonstrated in a number of application areas including scientific computing, signal and image processing, and logic simulation.

But there are many obstacles to full utilization of parallel computers. While the expectations are high, the performance of application programs on parallel

computers has not been impressive. It is true that some specific problems have been solved on specific computers with big gains in the time taken to solve the problem. There are no generally applicable methodologies to solve problems on parallel computers. One of the major problems is algorithm decomposition and task distribution in a parallel solution. Algorithms can be broadly classified into various classes based on the commonality of the approach taken [52,3]. Some of these classes (e.g., divide-and-conquer, dynamic programming, greedy, etc.) are well known. It occurred to us that instead of attacking the issue of decomposition and distribution in general, we should perhaps pick one class of algorithms and develop solutions to the decomposition and distribution problems. The class of algorithms we have chosen is divide-and-conquer.

One of the major problems in using parallel computers is the mismatch between the number of processes generated by the program and the number of processors available in the computer. In order to ensure that the program can be executed, we need to map the set of processes onto the set of processors based on some optimization criterion. However, this problem is known to be NP-complete. In this thesis, we focus on divide-and-conquer algorithms for MIMD computers. Divide-and-conquer algorithms form a large subclass of algorithms typically implemented on parallel computers. We propose an optimal strategy for implementing divide-and-conquer algorithms. This strategy bypasses the need for mapping. We also define an elegant measure of similarity that can be used for partitioning the set of processes. We consider the problem of obtaining a closed-form expression for the time complexity of a parallel program. It is very difficult to analytically derive an accurate expression for the time complexity. There are many factors like communi-

cation complexity, memory latency, etc., that can not be modeled precisely. Hence, we explore the possibility of using empirical methods. It is hoped that these methods would prove useful to experimenters involved in the study of parallel program behavior.

The rest of this dissertation is organized as follows. Chapter 2 consists of background work in the area and Chapter 3 contains an analysis of divide-and-conquer algorithms. In Chapter 4, we present various methods for implementing divide-and-conquer algorithms. Chapter 5 addresses the question of obtaining a closed-form expression for time complexity of a parallel program. Finally, Chapter 6 consists of conclusions and some ideas for future work.

CHAPTER 2. BACKGROUND

In this chapter, we introduce some of the terminology that is used in subsequent chapters. Previous work done in the area is briefly reviewed. The mapping problem is defined and some of the approaches to solving it are outlined. Also described are the simulated annealing method, cluster analysis and empirical analysis including the use of regression.

The Mapping Problem

After a program is written for a parallel computer the processing has to be distributed among different processors. This problem is called the mapping problem. The mapping problem is looked at from an idealized perspective of a real machine. But there are very complex issues that need to be addressed when actually executing a program and studying its behavior. We cease to be in the realm of elegant models and precise analysis. There are many interacting factors that influence program performance.

Many of the algorithms developed require a number of processors that is a function of the input size. Since in reality, the number of processors is usually bounded, there are serious problems when implementing such algorithms on actual parallel computers. Some researchers have addressed the problem of implementing

algorithms that require an unbounded number of processors by using the notion of *virtual processors* [56]. The programmer codes the algorithm assuming an unbounded number of processors is available. The program is preprocessed by *mapping software* that assigns multiple logical processors to a single physical processor. Typically a cost function is sought to be minimized. There are different versions of the mapping problem, but the issue of mismatch between number of processors needed and number of processors available is common to all of them. The mapping problem in full generality is NP-complete. Various heuristics have been used to solve the mapping problem. Two efforts at solving the mapping problem are worth considering. Next we will look at these efforts.

A software tool to solve the *mapping problem* for non-shared memory multiprocessors has been developed by Berman and Snyder [56,10]. They represent an instance of a parallel algorithm as a *communication graph* G_i whose nodes represent processes and whose edges represent communication links between processes. The parallel algorithm is then a family of communication graphs G_i, one for each problem instance. To represent the target multiprocessor, they use an undirected *Computation Graph* H in which the nodes are processors and the edges are data paths. The user inputs the algorithm and architecture using a graph description language. The mapping process can then be viewed as an embedding problem from G_i into H. This is accomplished using three transformations: *contraction, placement and routing.*

Contraction Module addresses the following problem: Given an undirected graph with m nodes, find a partition into at most $n \leq m$ groups such that a given cost function is minimized. Berman and Snyder tried using Simulated Annealing

[64] and Local Neighborhood Search [2] algorithms. They chose Local Neighborhood search. In their work on code partitioning Donnet and Skillicorn [30] report success with using Simulated Annealing . An overview of Simulated Annealing method is presented below.

Placement Module addresses the following problem: Given a graph G with at most n processes, find an embedding of G into a grid with n processors so that a given cost function is minimized. Berman and Snyder found both Simulated Annealing and Kernighan and Lin Algorithms [61] satisfactory.

Routing Module is an architecture dependent problem.

A major drawback with this approach is that for each instance of algorithm, the mapping problem needs to be solved. Berman and Snyder's approach requires that the problem size be known before execution. However, this may not be practical as in most cases the problem size is known only at run time.

Simulated Annealing Since simulated annealing is being used widely we have included an outline of the algorithm. This discussion is taken from [91]. Simulated Annealing belongs to a class of algorithms called Monte Carlo algorithms. Monte Carlo algorithms for optimization contain a random number generator associated with generation of points in the solution space.

Let $g(x)$ be defined on a finite set D. Assume that for each state d in D there exists a set $N(d)$ and an associated transition probability matrix $P_{dd'}$ such that $P_{dd'} > 0$ if and only if $d' \in N(d)$. Given any initial state, say $X_0 \in N(d)$, the proceeding states $X_1, X_2, \ldots$, are updated according to the following adaptive (annealing) algorithm.

$$X_{k+1} = \begin{cases} Y_k & \text{with probability } p_k \\ X_k & \text{with probability } 1 - p_k \end{cases}$$

where Y_k is chosen from $N(d)$ with probability

$P(Y_k = d' | X_k = d) = P_{dd'}$,

$p_k = e^{-[g(Y_k)-g(X_k)]^+/\sigma_k}$ [1]

and $\sigma_k, k = 1, 2, \ldots$ is a sequence (called a temperature schedule) of strictly positive numbers, such that $\sigma_1 \geq \sigma_2 \geq \ldots$ and $\sigma_k \to 0$ as $k \to \infty$.

The sequence $X_1, X_2, \ldots$, produced so far presents a nonhomogeneous discrete time Markov chain. The probability p_k to choose Y_k (next potential state) increases with σ_k (the higher the "temperature" is, the more likely is that a "hill climbing" move is accepted) and decreases with $[g(Y_k) - g(X_k)]^+$. The shape of the function g, the initial temperature, the rate of decrease of the temperature, and the number of iterations are all important parameters affecting the convergence and speed of convergence of the algorithm [78].

Let D^* denote the set of states in D where g achieves the global minimum. Hajek [78] discusses conditions under which

$\lim_{k \to \infty} P(X_k \in D^*) = 1$

and therefore the annealing algorithm converges to the global minimum.

Cluster Analysis

The use of clustering analysis for solving the mapping problem has been suggested by Pirktl [87] and Kim and Browne [62]. Clustering algorithms were first suggested by Pirktl and used recently by Kim and Browne. These algorithms are

[1] $\forall a, a^+ = a$ if $a > 0$, and $a^+ = 0$ otherwise.

commonly used in statistics to classify a sample into one of several different populations.

We use Cluster Analysis to solve the following problem.

> Given a set of objects, partition the set into groups such that variance within a group is small and variance between groups is large.

The basic objective in cluster analysis is to discover natural groupings of the objects. The inputs required are a similarity measure (or distance) and data from which similarities can be computed.

Following is an outline of Kim and Browne's work. A parallel computation can be represented by a directed acyclic graph $G_C = (N_C, E_C)$, where $N_C = \{n_1, n_2, \ldots, n_l\}$ is a set of schedulable units of computation to be executed, and E_C specifies scheduling constraints and data dependencies defined on N_C. A multiprocessor architecture can be represented by an undirected graph $G_P = (N_P, E_P)$, where $N_P = \{p_1, p_2, \ldots, p_m\}$ is a set of processors, and E_P specifies interconnection topology among the processors. The basic problem is to find a mapping of G_C onto G_P which minimizes schedule length (or makespan) defined as:

$\underset{1 \leq k \leq s}{Max} \sum_{i,j \in \phi_k} (comp_i + comm_{ij})$,

where $\phi = \{\phi_1, \phi_2, \ldots, \phi_s\}$ represents a set of paths from the root node to the leaf node in G_C, node n_j (assigned to processor $p_y \in N_P$ $(1 \leq y \leq m)$) is a direct descendant of node n_i (assigned to processor $p_x \in N_P$ $(1 \leq x \leq m)$) in G_C, $comp_i$ is computation time of n_i, and $comm_{ij}$ is communication time from n_i to n_j ($comm_{ij} = 0$, if $p_x = p_y$ or n_i has no direct descendants). An

optimal schedule is one which meets the criteria of minimum schedule length for a single parallel computation structure or the maximum total throughput for a set of simultaneously executing parallel computation structures.

One of the contributions of the work is to propose algorithms based on *linear clustering.* A linear cluster is a connected subgraph of a computation graph which is in the form of a linear list of schedulable units of computation. A computation graph is transformed into a *virtual architecture graph (VAG)* by linear clustering. The VAG may be transformed into another VAG by merging two or more linear clusters into one cluster.

After constructing a VAG which represents the optimal multiprocessor architecture for a given computation graph, they then find an optimal mapping of the VAG onto a *physical architecture graph (PAG)* which represents the target architecture.

Empirical Analysis

In the study of various systems we want to comment on interrelationships of various factors. We built a model to facilitate such a study. By a model of a system we mean a formal statement about interactions of different factors that contribute to understanding of the system.

There are several approaches to model building. Sometimes we can combine "known theory" with logical reasoning, to obtain the model. We start with some ground rules and derive the model. This is called conceptual modeling. The model can be deterministic or stochastic (probabilistic). In case of a deterministic model the behavior of the system is completely predictable, whereas in case of a stochastic

model we need to worry about some random fluctuations.

At the other extreme the system may be too complex or too difficult to model conceptually. Then we try to find the relationships between inputs and outputs based entirely on past behavior of the system. To put it in plain English we consider only past data and determine the relationships. This approach is called a purely empirical approach. The model we build can be used to predict future behavior of the system. But in reality a pure empirical approach is rarely used. We almost always have prior information about the system and we try to make use of that information. When we talk of an empirical model we do not mean a pure empirical model but we refer to empirical modeling assisted by prior information.

If we can explain system conceptually then there would be no need for an empirical approach. But due to a variety of reasons like lack of clear knowledge of the underlying mechanism and relationships, the complexity of complete specification, etc., it is often not possible to use the conceptual approach. When dealing with parallel programs we face such a situation.

Our approach to estimating the complexity of parallel programs involves measuring the factors that contribute to time complexity and the actual time taken by the parallel program. We then obtain an expression for parallel program complexity based on that data. We use a statistical technique called "Regression Analysis" to achieve this. The discussion on regression analysis is taken from [20].

Another important issue that we have to address when we want to execute a parallel program on a parallel computer is how to assign different processes to various processors. This is a very complex theoretical problem. There is no generally applicable fast (polynomial time) algorithm to solve the problem.

Regression Analysis The past twenty years have seen a great surge of activity in the general area of model fitting. Several factors have contributed to this, not the least of which is the penetration and extensive adoption of the computers in statistical work. The fitting of linear regression models by least squares is undoubtedly the most widely used modeling procedure.

The elements that determine a regression equation are the observations, the variables, and the model assumptions.

Notations We are concerned with the general regression model

$$Y = X\beta + \varepsilon, \tag{2.1}$$

where

Y is an $n \times 1$ vector of response or dependent variable;

X is an $n \times k (n > k)$ matrix of predictors (explanatory variables, regressors; carriers, factors, etc.), possibly including one constant predictor;

β is a $k \times 1$ vector of unknown coefficients (parameters) to be estimated; and

ε is an $n \times 1$ vector of random disturbances.

The identity matrix, the null matrix, and the vector of ones are denoted by I, **0**, and **1**, respectively. The ith unit vector (a vector with one in ith position and zero elsewhere) is denoted by u_i. We use M^T, M^{-1}, M^{-T}, $rank(M)$, $trace(M)$, $det(M)$, and $\| M \|$ to denote the transpose, the inverse, the transpose of the inverse (or the inverse of the transpose), the rank, the trace, the determinant, and the spectral norm of the matrix M, respectively.

We use y_i, x_i^T, $i = 1, 2, \cdots, k$, to denote the i^{th} element of Y and the i^{th} row of X, respectively, and $X_j, j = 1, 2, \cdots, k$, to denote the j^{th} column of X. By the

i^{th} observation(case) we mean the row vector $(x_i^T : y_i)$, that is, the i^th row of the augmented matrix

$$Z = (X : Y) \tag{2.2}$$

The notation "(i)" or "$[j]$" written as a subscript to a quantity is used to indicate the omission of the i^{th} observation or the j^{th} variable, respectively. Thus, for example, $X_{(i)}$ is the matrix X with the i^{th} row omitted, $X_{[j]}$ is the matrix X with the j^{th} column omitted, and $\hat{\beta_{(i)}}$ is the vector of estimated parameters when the i^{th} observation is left out. We also use the symbols $e_{Y.X}$ (or just e, for simplicity), $e_{Y.X_{[j]}}$, and $e_{X_j.X_{[j]}}$ to denote the vectors of residuals when Y is regressed on X, Y is regressed on $X_{[j]}$, and X_j is regressed on $X_{[j]}$, respectively.

Finally, the notations $E(.)$,$Var(.)$, and $Cor(.,.)$ are used to denote the expected value, the variance, the covariance, and the correlation coefficient(s) of the random variables(s) indicated in the parentheses, respectively.

Standard Estimation Results in Least Squares The least squares estimator of β is obtained by minimizing

$$(Y - X\beta)^T(Y - X\beta) \tag{2.3}$$

Taking the derivative of the expression 2.3 with respect to β yields the normal equations

$$(X^TX)\beta = X^TY \tag{2.4}$$

The system of linear equations 2.4 has a unique solution if and only if $(X^TX)^{-1}$

exists. In this case, premultiplying the equation 2.4 gives the least squares estimator of β, that is

$$\hat{\beta} = (X^T X)^{-1} X^T Y \tag{2.5}$$

In the Section 2 we give the assumptions on which several of the least squares results are based. If these assumptions hold, the least squares theory provides us with the following well-known results.

1. The $k \times 1$ vector $\hat{\beta}$ has the following properties:

   (a)

   $$E(\hat{\beta}) = \beta \tag{2.6}$$

   That is, $\hat{\beta}$ is an unbiased estimator for β.

   (b) $\hat{\beta}$ is the best linear unbiased estimator(BLUE) for β, that is among the class of linear unbiased estimators, $\hat{\beta}$ has the smallest variance. The variance of $\hat{\beta}$ is

   $$Var(\hat{\beta}) = \sigma^2 (X^T X)^{-1} \tag{2.7}$$

   (c)

   $$\hat{\beta} = N_k(\beta, \sigma^2 (X^T X)^{-1}) \tag{2.8}$$

   where $N_k(\mu, \Sigma)$ denotes a k-dimensional multivariate normal distribution with mean μ (a $k \times 1$ vector) and variance Σ (a $k \times k$ matrix).

2. The $n \times 1$ vector of fitted (predicted) values

   $$\hat{Y} = X\hat{\beta} = X(X^T X)^{-1} X^T Y = PY \tag{2.9}$$

where

$$P = X(X^T X)^{-1} X^T \tag{2.10}$$

has the following properties:

(a)

$$E(\hat{Y}) = X\beta \tag{2.11}$$

(b)

$$Var(\hat{Y}) = \sigma^2 P \tag{2.12}$$

(c)

$$\hat{Y} \sim N_n(X\beta, \sigma^2 P). \tag{2.13}$$

3. The $n \times 1$ vector of ordinary residuals

$$e = Y - \hat{Y} = Y - PY = (I - P)Y \tag{2.14}$$

has the following properties:

(a)

$$E(e) = 0. \tag{2.15}$$

(b)

$$Var(e) = \sigma^2(I - P). \tag{2.16}$$

(c)

$$e \sim N_n(0, \sigma^2(I - P)). \tag{2.17}$$

(d)

$$\frac{e^T e}{\sigma^2} \sim \chi^2_{(n-k)}, \tag{2.18}$$

where $\chi^2_{(n-k)}$ denotes a χ^2 distribution with $n-k$ degrees of freedom (d.f.) and $e^T e$ is the residual sum of squares.

4. An unbiased estimator of σ^2 is given by

$$\hat{\sigma^2} = \frac{e^T e}{n-k} = \frac{Y^T(I-P)Y}{n-k}. \tag{2.19}$$

Assumptions The least squares results and the statistical analysis based on them require the following assumptions:

Linearity Assumption This assumption is implicit in the definition of model 2.1, which says that each observed response value y_i can be written as a linear function of the i_{th} row of X, x_i^T, that is,

$$y_i = x_i^T \beta + \varepsilon_i, i = 1, 2, \cdots, n. \tag{2.20}$$

Computational Assumption In order to find a unique estimate of β it is necessary that $(X^T X)^{-1}$ exist, or equivalently

$$rank(X) = k. \tag{2.21}$$

Distributional Assumptions The statistical analyses based on least squares (e.g., the t-tests, the F-test, etc.) assume that

1.

$$X is measured without errors, \tag{2.22}$$

2.

$$\varepsilon_i does not depend on x_i^T, i = 1, 2, \cdots, n, and \tag{2.23}$$

3.

$$\varepsilon \sim N_n(0, \sigma^2 I). \tag{2.24}$$

The Implicit Assumption All observations are equally reliable and should have an equal role in determining the least squares results and influencing conclusions.

It is important to check the validity of these assumptions before drawing conclusions from an analysis. Not all of these assumptions, however, are required in all situations. For example, for 2.11 to be valid, the following assumptions must hold: 2.20, 2.21, 2.22, and part of 2.24, that is, $E(\varepsilon) = 0$. On the other hand, for 2.7 to be correct, assumption 2.23, in addition to the above assumptions, must hold.

Iterative Regression Process The standard estimation results given above are merely the start of regression analysis. Regression analysis should be viewed as a set of data analytic techniques used to study the complex interrelationships that may exist among variables in a given environment. It is a dynamic iterative process; one in which an analyst starts with a model and a set of assumptions and modifies them in the light of data as the analysis proceeds. Several iterations may be necessary before an analyst arrives at a model that satisfactorily fits the observed data.

CHAPTER 3. ANALYSIS OF DIVIDE-AND-CONQUER ALGORITHMS

In this chapter, we develop some results about time complexity of divide-and-conquer algorithms on parallel machines. The empirical models that we obtain for the parallel time complexity of divide-and-conquer algorithms will be based on the theorems that we establish in this chapter. This chapter provides the analytical framework for model building.

In Section 3, we provide an expression for time complexity of parallel divide-and-conquer algorithms. We define two functions called divide-and-combine function and communication function to account for time spent in dividing a problem and combining solutions of subproblems, and time spent in communication. Often communication time is ignored in the analysis of parallel algorithms. But in reality, communication time is significant. Recently Leiserson and Maggs [67] have introduced a theoretical model called the D-RAM to account for interprocessor communication time. We observe that, more than individual complexities of divide-and-combine function and communication function, what really matters is the nature of the sum of these two functions. We consider the situations where the complexity of sum of divide-and-combine function and communication function is (1)$O(1)$ (*constant*) (2)$O(\log n)$ and (3)$O(n)$ (*linear in* n). We then prove a theo-

rem that helps us comment on the complexity of divide-and-combine function and communication function in case of $O(\log n)$ parallel algorithms.

Divide and Conquer Paradigm

The time complexity of a divide-and-conquer algorithm is of the form:

$$T(n) = \begin{cases} T(n/k) + f_1(n) + f_2(n) + c & for \quad n > x \\ 1 & for \quad n <= x \end{cases}$$

Where n is a positive integer, f_1 and f_2 are positive functions, k and x are positive integer constants, c is a real constant. Many of the commonly used divide-and-conquer algorithms satisfy above definition. Next we will consider an interpretation of $T(n)$. We could assume that the problem at hand is subdivided into several problems each of size $\frac{n}{k}$. Solving each subproblem takes $T(\frac{n}{k})$ time. We will suppose that all the subproblems are solved in parallel. So $T(\frac{n}{k})$ is time to solve subproblems. Typically a divide-and-conquer algorithm has three phases. They are

1. Divide the problem
2. Solve subproblems and
3. Combine the subproblem solutions.

Phase 2 has three stages. They are

a. send data for subproblems,

b. wait for solution of subproblems and

c. receive results.

We have seen that $T(\frac{n}{k})$ is the time for solving subproblems. We will assume that $f_1(n)$ accounts for time for dividing the problem and the time for solving sub-

problems. We will call f_1 divide-and-combine function. We will assume that each subproblem is solved on a different processor. This requires that subproblems be transmitted to the processors solving subproblems and the results be returned from processors solving subproblems. We have to account for this communication. We will use $f_2(n)$ to account for time for communication. We will call $f_2(n)$ communication function. Depending on the physical characteristics of the computer $f_2(n)$ could mean the time for message passing or delay due to memory/bus contention or a combination of both.

We will consider different forms of functions for $f_1(n)$ and $f_2(n)$ and comment on the time complexity. First we will assume that $f_1(n)$ and $f_2(n)$ are constant functions. Next we will assume that both $f_1(n)$ and $f_2(n)$ are linear functions in n.

Divide-and-Combine and Communication Times are Constant

Let us suppose $f_1(n) = c_1$ and $f_2(n) = c_2$. Then we have

Theorem 1 *Any divide-and-conquer algorithm with a constant divide-and-combine function and a constant communication function can be solved in* $O(\log n)$ *parallel time.*

Proof:

$$T(n) = \begin{cases} T(n/k) + c_1 + c_2 + c & for \quad n > x \\ 1 & for \quad n <= x \end{cases}$$

This can be written as

$$T(n) = \begin{cases} T(n/k) + c & for \quad n > x \\ 1 & for \quad n <= x \end{cases}$$

Without loss of generality let $n = 2^l$ and $k = 2^m$ where l and m are positive integers.

$$\begin{aligned} T(n) &= T(n/k) + c \\ &= T(2^{l-m}) + c \\ &= T(2^{l-2m}) + 2c \\ &= T(2^{l-3m}) + 3c \\ &\vdots \\ &= T(2^{l-(\frac{l}{m})m}) + (\frac{l}{m})c \\ &= 1 + \frac{c}{m}\log n \\ &= O(\log n) \end{aligned}$$

Divide-and-Combine and Communication Times are Linear

Let us suppose $f_1(n) = c_1 n + a_1, c_1 > 0$ and $f_2(n) = c_2 n + a_2, c_2 > 0$. Then we have

Theorem 2 *Any divide-and-conquer algorithm with both divide-and-combine function and communication function linear in n can be solved in $O(n)$ parallel time.*

Proof:

$$T(n) = \begin{cases} T(n/k) + c_1 n + c_2 n + a_1 + a_2 + a_3 & for \quad n > x \\ 1 & for \quad n <= x \end{cases}$$

Without loss of generality let $n = 2^l$ and $k = 2^m$ where l and m are positive integers. Let$k_1 = c_1 + c_2$and$k_2 = a_1 + a_2 + a_3$

$$
\begin{aligned}
T(n) &= T(\frac{n}{k}) + k_1 n + k_2 \\
&= T(\frac{2^l}{2^m}) + k_1 2^l + k_2 \\
&= T(2^{l-m}) + k_1 2^l + k_2 \\
&= T(2^{l-2m}) + k_1 2^{l-m} + k_2 + k_1 2^l + k_2 \\
&= T(2^{l-2m}) + k_1 [2^{l-m} + 2^l] + 2k_2 \\
&= T(2^{l-3m}) + k_1 2^l [2^{-2m} + 2^{-m} + 2^0] + 3k_2 \\
&\vdots \\
&= T(2^{l-\frac{l}{m}m}) + k_1 2^l [2^{-(\frac{l}{m}-1)m} + \ldots + 2^{-2m} + 2^{-m} + 2^0] + \frac{l}{m} k_2 \\
&= T(1) + k_1 n [(\frac{n-1}{n})(\frac{k}{k-1})] + \frac{k_2}{\log k} \log n \\
&= T(1) + k_1 (n-1)(\frac{k}{k-1}) + \frac{k_2}{\log k} \log n \\
&= O(n)
\end{aligned}
$$

$O(\log n)$ Parallel Divide-and-Conquer Algorithms

In this section we identify some properties of $O(\log n)$ parallel algorithms. In particular we will comment on the complexity of divide-and-combine and communication functions. We have

Theorem 3 *For any divide-and-conquer algorithm with parallel time complexity* $O(\log n)$ *the sum of divide-and-combine function and communication function is* $O(\log n)$.

Proof: Let us suppose $g(n) = f_1(n) + f_2(n)$.

$$T(n) = \begin{cases} T(n/k) + g(n) + c & for \quad n > x \\ 1 & for \quad n <= x \end{cases}$$

Without loss of generality let $n = 2^l$ and $k = 2^m$ where l and m are positive integers.

$$\begin{aligned}
T(n) &= T(\tfrac{n}{k}) + g(n) + c \\
&= T(\frac{2^l}{2^m}) + g(2^l) + c \\
&= T(2^{l-m}) + g(2^l) + c \\
&= T(2^{l-2m}) + g(2^{l-m}) + c + g(2^l) + c \\
&= T(2^{l-2m}) + g(2^{l-m}) + g(2^l) + 2c \\
&= T(2^{l-3m}) + [g(2^{l-2m}) + g(2^{l-m}) + g(2^l)] + 3c \\
&\vdots \\
&= T(2^{l-\frac{l}{m}m}) + [g(2^{-(\frac{l}{m}-1)m}) + \ldots + g(2^{l-2m}) + g(2^{l-m}) + g(2^l)] + \frac{l}{m}c \\
&= T(1) + [g(2^{-(\frac{l}{m}-1)m}) + \ldots + g(2^{l-2m}) + g(2^{l-m}) + g(2^l)] + \frac{c}{\log k}\log n \\
&= O(\log n) + [g(2^{-(\frac{l}{m}-1)m}) + \ldots + g(2^{l-2m}) + g(2^{l-m}) + g(2^l)]
\end{aligned}$$

We know that $T(n) = O(\log n)$ and

$T(n) = O(\log n) + [g(2^{-(\frac{l}{m}-1)m}) + \ldots + g(2^{l-2m}) + g(2^{l-m}) + g(2^l)]$

Therefore,

$[g(2^{-(\frac{l}{m}-1)m}) + \ldots + g(2^{l-2m}) + g(2^{l-m}) + g(2^l)] = O(\log n)$

$g(n)$ is a positive function.

Table 3.1: Time Complexity of Divideand-Conquer Algorithms

Communication Complexity	Divide-and-Combine Complexity		
	c	$O(logn)$	$O(n)$
c	c	$O(logn)$	$O(n)$
$O(logn)$	$O(logn)$	$O(logn)$	$O(n)$
$O(n)$	$O(n)$	$O(n)$	$O(n)$

Hence, $g(n) = O(\log n)$ *q.e.d*

It follows from **Theorem 3**:

Theorem 4 *Any divide-and-conquer algorithm with both divide-and-combine function and communication function of complexity* $O(\log n)$ *can be solved in* $O(\log n)$ *parallel time.*

Table 3.1 summarizes the results obtained in this Chapter.

CHAPTER 4. IMPLEMENTATION ON MIMD MACHINES

In this chapter, we consider a subclass of algorithms, viz., divide-and-conquer, and offer a strategy for implementing them that eliminates the need for solving the mapping problem. The divide-and-conquer paradigm is a widely-used technique for algorithm design on sequential computers. It has been used in a variety of areas, for example, graph theory, matrix computations, computational geometry, FFT, etc. [52,3]. Philip Nelson [56,83] considers it to be one of the programming paradigms for parallel computers. Horowitz and Zorat [53] discuss special architectures for divide-and-conquer computers. Our focus will be on designing a methodology for implementing divide-and-conquer algorithms on a bounded number of processors.

Implementation on Bounded Number of Processors

This section examines the suitability of the divide-and-conquer paradigm from a practical point of view. The focus is on implementing concrete algorithms on parallel computers with a bounded number of processors, and studying the effect of this paradigm on the performance of the algorithms. A design technique for a divide-and-conquer paradigm is developed and analyzed for parallel computers with a bounded number of processors.

In Subsection 4 we present the divide-and-conquer paradigm and develop a

technique to ensure that an algorithm does not need any more processors than are available. Our technique does not assume knowledge of the problem size before execution. In Subsection 4 we modify our technique to minimize idle time for processors. Results based on experiments conducted at Argonne National Laboratories lead us to believe that our techniques for divide-and-conquer algorithms are quite useful in practice.

Divide-and-conquer paradigm

In this section we first describe the outline of a divide-and-conquer algorithm. We then suggest modifications to this outline that are suitable for parallel computers with a bounded number of processors.

The following is an outline of a divide-and-conquer algorithm:

```
basic divide-and-conquer
begin
    if problem size is small enough
       then
          solve the problem right away (*)
       else
           begin (**)
               subdivide the problem
                    into subproblems
               solve the subproblems
               combine the solutions
           end;
```

```
end.
```

In trying to obtain a parallel version of such a divide-and-conquer algorithm, the following problems have to be dealt with. Ideally, one would like to assign a distinct physical processor for each process (i.e., subproblem) of the divide-and-conquer strategy. However, this is not always possible as we have only a fixed number of processors. Our strategy in designing a parallel version is to ensure that the number of processes spawned by the algorithm does not exceed the number of processors available. The key issue in matching the number of subproblems to the number of available processors is to determine the right problem size that is "small enough" to serve as a cutoff point for executing parts (*) and (**) of the basic divide-and-conquer algorithm. In other words, the problem is similar to that of determining the optimal grain size on the multiprocessor.

We use the following notation in our analysis. The number of subproblems generated at each stage by the divide-and-conquer strategy is K. The problem size is N and the number of available processors is M. In designing a parallel version, the goal is to determine a bound x such that when the problem size is $\leq x$ then no subproblems are created and the problem is solved right away. An outline of the parallel version is as follows:

```
parallel divide-and-conquer-1
begin
    if problem size  <=  x
       then
           solve the problem right away
       else
```

```
        begin
            subdivide the problem
               into K subproblems
            solve the subproblems
               in parallel
            combine the solutions
        end;
end.
```

To complete the analysis, we need to derive a formula for determining x. Since the number of subproblems spawned at each stage of the "divide" process is K, the process graph is a K-ary tree. Processes that correspond to leaves of the process graph do not generate any subproblems. Hence, their size should be bounded by x.

$x = \lceil N/K^y \rceil$ where y is the height of the process tree.

We want the largest value y such that,

$1 + K + \cdots + K^y \leq M$

$(K^{y+1} - 1)/(K - 1) \leq M$

$y = \lfloor \log_K(M(K-1)+1) - 1 \rfloor$

In the above analysis, problem size was assumed to be a function of one variable, N. It is easy to extend this analysis for the case where problem size is a function of many variables. Suppose problem size is a vector $\tilde{N} = (n_1, n_2, \cdots, n_p)$. Suppose the i^{th} variable has k_i subdivisions for the purpose of subdividing the problem. Then there are $K = \prod_{i=1}^{p} k_i$ subproblems. The derivation for $\tilde{x}$ is derived in a manner quite similar to that done above.

$\tilde{x} = (\lceil n_1/K^y \rceil, \lceil n_2/K^y \rceil, \cdots, \lceil n_p/K^y \rceil)$

Where $y = \lfloor \log_K(M(K-1)+1) - 1 \rfloor$

Can we do better?

This section improves upon processor utilization of the basic technique developed in the previous section. Between the time a processor spawns processes for solving subproblems and begins receiving solutions for subproblems, the processor is idle. We would like the processor to do some useful computation during this period. Next we describe how this can be done.

We could have a processor solve a fraction (say $p, 0 \leq p \leq 1$) of the problem. The processor would create subproblems to solve $(1-p)^{th}$ of the problem. We would like to choose p such that the time to solve p^{th} of the problem should be about the same as the time to solve the subproblems.

If T_P and T_S denote the parallel and sequential time complexity of the algorithms respectively, p may be obtained by solving the following equation.

$T_P(\frac{(1-p)N}{K}) = T_S(pN)$

Often it is difficult to obtain closed form expressions for T_S and T_P. We can use statistical techniques for estimating p [82].

The value of x determined in the previous section now changes to:

$x = \lceil (1-p)^{y+1} N/K^y \rceil$

where $y = \lfloor \log_K(M(K-1)+1) - 1 \rfloor$

The revised algorithm would be:

```
parallel divide-and-conquer-2
begin
    if problem size  <= x
```

```
        then
            solve the problem right away
        else
             begin
                 parbegin
                 solve K subproblems of
                       (1-p)th problem
                 solve p th problem
                 parend;
                 combine the solutions
             end;
    end.
```

Next we will consider an interesting special case when $p = L/K$ where L is an integer between 0 and K. Then each process need only spawn processes to solve $(K - L)$ subproblems. The revised value for x is:

$x = \lceil N/(K-L)^y \rceil$ where y is the height of the process tree.

We want the largest value y such that,

$1 + (K-L) + \cdots + (K-L)^y \leq M$

$((K-L)^{y+1} - 1)/(K-L-1) \leq M$

$y = \lfloor \log_{K-L}(M(K-L-1)+1) - 1 \rfloor$

The revised algorithm would be:

```
parallel divide-and-conquer-3
begin
    if problem size  <= x
```

```
        then
            solve the problem right away
        else
             begin
                 parbegin
                 have the (K-L) subproblems
                                solved;
                 solve L subproblems
                 parend;
                 combine the solutions
             end;
end.
```

An example

We have implemented some of the suggestions made in the previous section. We have used the mergesort algorithm for implementation. This algorithm was implemented on Intel iPSC (32 node) located at Argonne National Laboratories. The algorithm is a typical divide-and-conquer algorithm. The list to be sorted is split into two sublists and each one is sorted using merge sort. Then the sorted lists are merged. Details of Intel iPSC computer are provided in Appendix A. Techniques for programming iPSC are given in Appendix B.

For our implementation node 0 acted as the controller. Node 0 was responsible for generating data and collecting statistics. Timers on iPSC nodes give time in milliseconds. The clocks are updated every five seconds. Clocks on different nodes

Table 4.1: p vs. time

p	time
0.0	1517145
0.1	948730
0.2	923790
0.3	1052955
0.4	1256215
0.5	1492670
0.6	1734125
0.7	1978870
0.8	2225445
0.9	2474710

are not synchronized.

The program was executed on cube varying in dimension from one to five. Problem size was varied from 100 to 10000 in steps of 100. We have used the technique presented in Section 4. This technique requires a value for p. Here p is the proportion of problem to be solved sequentially. On each processor (except leaf nodes) p^{th} of the problem is solved on the same node and $(1-p)^{th}$ of the problem is divided to be solved in parallel. There in no easily obtainable closed form expression for p. We have executed the program for values of p ranging from 0.0 to 0.9 in steps of 0.1. We have presented aggregates of time for each value of p in table 4.1. Figure 4.1 contains a graph depicting the observations and predicted values is enclosed.

We fit a curve of degree to these data. The function is:

$$Time = f(p) = 1241091.29 - 1219076.97p + 3045469.70p^2$$

To obtain p that minimizes $f(p)$:

$$\frac{df}{dp} = -1219076.97 + 2 * 3045469.70p = 0$$

$$p_{opt} = 0.200145$$

What that means is that the time solve $1/5^{th}$ of the problem sequentially is about same as the time to solve $4/5^{th}$ of the problem in parallel. However, we need to exercise caution in interpreting the results since the dimension of the cube is small.

A General Method

As defined in a previous chapter the mapping problem involves obtaining a mapping of processes in a program onto processors. There are two stages in solving the mapping problem. One involves partitioning the processes and the second stage involves assigning each partition to a processor. First we will explain our approach to partitioning of processes. We need to ensure that the number of partitions is less than or equal to number of processors available.

A new measure of similarity among processes

As stated in a previous chapter several researchers have considered the problem of code partitioning. One serious problem with the solutions offered is that sole criterion for partitioning the set of processes is the communication traffic between the processes. We propose consideration of another important criterion. That is the discordance of CPU requirement among processes within a partition/cluster. What this means is that we should try to partition processes such that within a partition

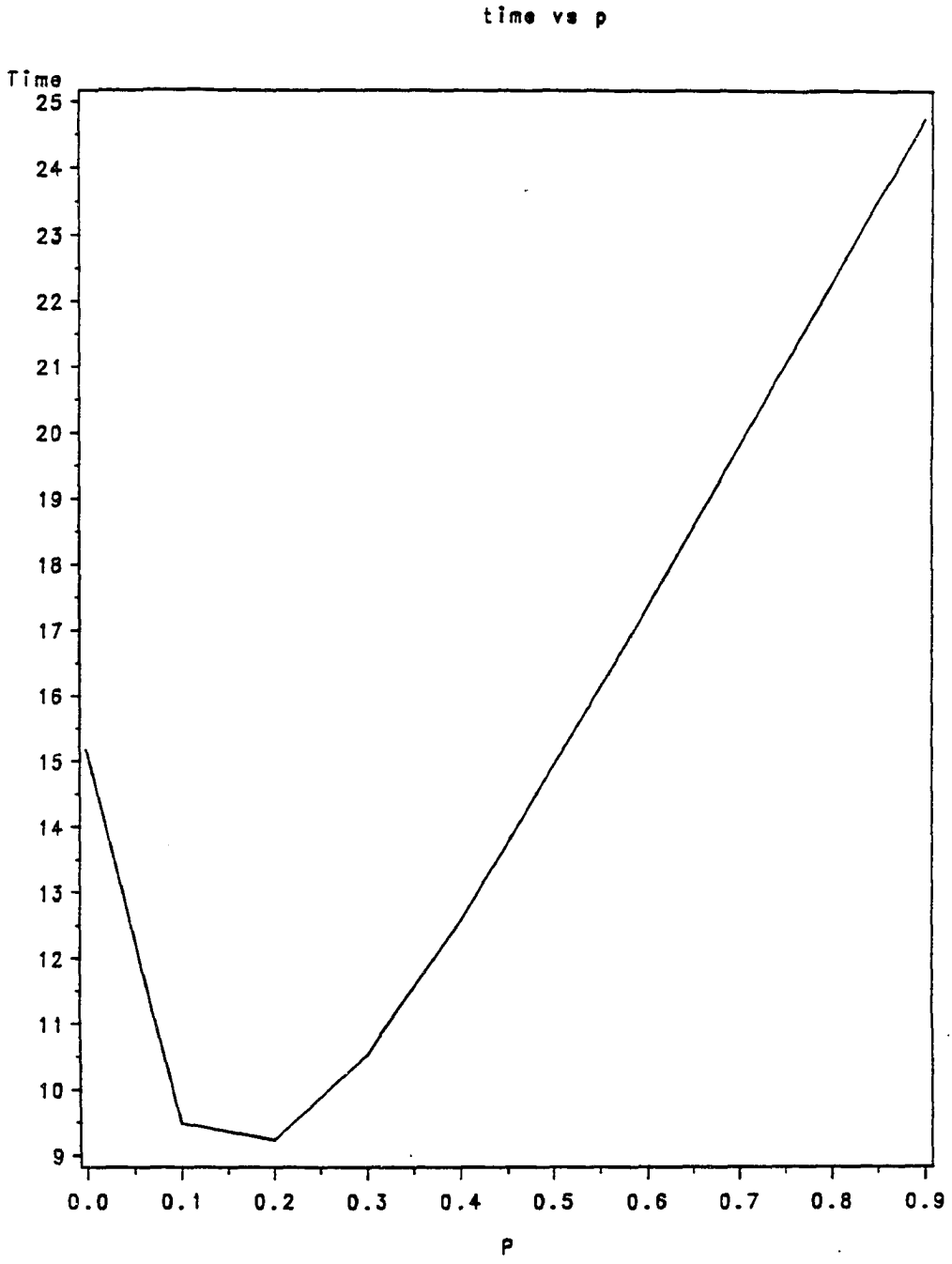

Figure 4.1: p Vs. Time

CPU requirement for processes within a partition should be as disjoint as possible. We need to consider how to minimize the wait for CPU within a partition. Hence we have two important factors to consider when partitioning tasks/processes. They are:

1. Communication Traffic
2. Discordance of CPU requirement

We will next develop a measure of similarity based on these criteria. For each pair of distinct processes i and j, we define a similarity measure based on x_{ij} and y_{ij}. Here x_{ij} indicates the amount of communication traffic between process i and process j. This could be in bytes or some other units. y_{ij} indicates the proportion of time process i and process j do not need the CPU at the same time. $y_{ij} \in [0,1]$. If there are P processes in the program then we have $N = \frac{P(P-1)}{2}$ pairs of (x_{ij}, y_{ij})'s. Now we will define some statistics based on (x_{ij}, y_{ij})'s.

$$
\begin{aligned}
\bar{x} &= \sum_{i<j=1}^{M} x_{ij}/N \\
\bar{y} &= \sum_{i<j=1}^{M} y_{ij}/N \\
s_{xx}^2 &= \sum_{i<j=1}^{M} (x_{ij} - \bar{x})^2/N \\
s_{yy}^2 &= \sum_{i<j=1}^{M} (y_{ij} - \bar{y})^2/N \\
s_{xy} &= \sum_{i<j=1}^{M} (x_{ij} - \bar{x})(y_{ij} - \bar{y})/N \\
s_{yx} &= s_{xy}
\end{aligned}
$$

$$r_{xy} = \frac{s_{xy}}{s_{xx}s_{yy}}$$
$$r_{yx} = r_{xy}$$

We define two new variables

$$x'_{ij} = \frac{(x_{ij} - \bar{x})}{s_{xx}}$$
$$y'_{ij} = \frac{(y_{ij} - \bar{y})}{s_{yy}}$$

By subtracting $\bar{x}, \bar{y}$ from x_{ij} and y_{ij} we are making x'_{ij} and y'_{ij} location independent. By dividing by s_{xx} and s_{yy} we eliminate units. x'_{ij} and y'_{ij} are called standardized variables.

$$E(\acute{x}) = E(\acute{y}) = 0$$

$$V(\acute{x}) = V(\acute{y}) = 1$$

Removing the units is necessary because we are going to combine x's and y's to define a similarity measure. It is possible that there is a correlation between x's and y's. If we do not eliminate the effect of this correlation the similarity measure will be distorted. The similarity measure that we propose is:

Similarity between processes i and j is defined as

$$S(i,j) = \begin{pmatrix} x'_{ij} \\ y'_{ij} \end{pmatrix}' \begin{pmatrix} s_{xx} & s_{xy} \\ s_{yx} & s_{yy} \end{pmatrix}^{-1} \begin{pmatrix} x'_{ij} \\ y'_{ij} \end{pmatrix}$$
$$= \begin{pmatrix} x'_{ij} \\ y'_{ij} \end{pmatrix}' \begin{pmatrix} 1 & r_{xy} \\ r_{xy} & 1 \end{pmatrix}^{-1} \begin{pmatrix} x'_{ij} \\ y'_{ij} \end{pmatrix}$$

$$= \begin{pmatrix} x'_{ij} \\ y'_{ij} \end{pmatrix}' \frac{1}{1-r_{xy}^2} \begin{pmatrix} 1 & -r_{xy} \\ -r_{xy} & 1 \end{pmatrix} \begin{pmatrix} x'_{ij} \\ y'_{ij} \end{pmatrix}$$

$$= \frac{1}{1-r_{xy}^2}(x_{ij}^2 + y_{ij}^2 - 2r_{xy}x_{ij}y_{ij})$$

For divide-and-conquer problems it is easy to estimate x_{ij}'s and y_{ij}'s. We will explain the algorithm in the next section. In general it is easy to determine x_{ij}'s but it is not easy to determine y_{ij}'s. One way to estimate y_{ij}'s would be to check the status of different processes every few time units. We need to set up a counter for every process pair. If we find that process i and process j are both running or one of them is running and the other is waiting for CPU then increment the counter for (i, j). At the end of processing counter values need to be divided by total number of checks made. This would give us estimates for y_{ij}'s. The estimates are not precise but seem to be ok.

The method we have outlined can be easily extended to techniques that need to use more than two factors.

Computing the measure for divide-and-conquer problems

The Communication Graph (or Process Graph) of a divide-and-conquer algorithm is a K-ary tree, where K indicates the number of subproblems. The amount of communication between a node and its child depends on the problem and can easily be estimated. But how do we estimate CPU Discordance? Observe that nodes at the same height are expected to be executing at the same time. That means CPU Discordance between nodes at the same level is very low. In general CPU Discordance between two nodes would depend on the difference in the heights

of the nodes. We also know that a node and any of it's descendants would never need the CPU at the same time. That means the CPU Discordance between a node and any of its descendents is very high. Based on these observations, we state the following formula for determining CPU Discordance y_{ij} between any two processes i and j.

$$y_{ij} = \begin{cases} 0 & if\ i = j \\ height\ of\ the\ Process\ Graph & if\ i\ is\ a\ descendent\ of\ j\ or \\ & j\ is\ a\ descendent\ of\ i \\ |\ height(i)\ -\ height(j)\ | & Otherwise \end{cases}$$

Generally applicable methodology

Using the measure of similarity that we have just developed we can now provide a widely applicable methodology for implementing divide-and-conquer algorithms on MIMD machines.

We will use the following notation:

N: Problem Size

P: Number of Processes in the Process Graph

K: Number of Subproblems at each node

M: Number of processors in the machine

Following is the sequence of steps involved in implementing a divide-and-conquer algorithm on a MIMD machine.

1. If the algorithm can be adapted to the technique used for implementing merge-sort then implement the algorithm with various values for p and obtain a good

estimate of p_{opt}. But there are many divide-and-conquer algorithms that can not be adapted to this method. Typically the problem lies in combining the solutions of subproblems. In such cases the subsequent steps need to be followed. One should keep in mind that the partitioning is done at run time.

2. Read the value of N, the Problem Size

3. On some machines there are limits on total number of processes that can be generated. Or for reasons of efficiency we may want to limit the number of processes generated. Suppose the limit on the number of processes in the system is T. If the number of processes required for the Problem of Size N is greater than T then determine a value of x according to the formula developed in Section 4. Else use x as stated in the algorithm.

4. Compute $S(i,j)$ for $1 <= i,j <= P$. That is, compute the Measure of Similarity for each pair of the processes.

5. We need to map P processes onto M processors. In other words we want to partition P into M groups (or classes or clusters). Optimal partitioning problem is NP-Complete. We have tried using Cluster Analysis and Simulated Annealing methods for obtaining good partitioning. These two methods are approximate methods. They do not guarantee optimal solutions. There is no clear choice. Suppose the partitions are $C_1, C_2, \cdots, C_M$.

6. Assign the partitions $C_1, C_2, \cdots, C_M$ to M processors. The algorithm used for the assignment is machine dependent.

At this point we have assigned each process to one of the M processors. Information about this assignment is available at each processor.

7. Next we will execute the following algorithm:

```
parallel divide-and-conquer-4
begin
    if problem size  <=  x
       then
           solve the problem right away
       else
            begin
                subdivide the problem
                   into K subproblems
                assign the subproblems
                   appropriate processors
                combine the solutions
            end;
end.
```

Summary

In this chapter we looked in detail at the question of implementing divide-and-conquer algorithms on MIMD computers. We have to take into consideration the fact that the number of processors in a computers is fixed. In other words the computer can not expand to fit the process graph of a problem. We have developed

techniques for implementing divide-and-conquer algorithms on MIMD computers. For some divide-and-conquer algorithms we can bypass the process of mapping. This method is illustrated by implementing mergesort algorithm. But there are many divide-and-conquer algorithms that are not amenable to this method. Then we propose a generally applicable technique for implementing divide-and-conquer algorithms. The key to this techniques is the measure of similarity among processes that we define in this chapter. A notable aspect of this measure is that it considers not only the communication traffic between processes but also the contention for central processing unit(CPU) among processes that are assigned to the same CPU.

CHAPTER 5. PERFORMANCE MODELING

Parallel Program Complexity

In this section we address the following question:

> How much time would a parallel program solving a problem of size N be expected to take when executed on an MIMD machine with K processors?

We are interested in studying the behavior of parallel programs when executed on an MIMD machine. We represent a parallel program by a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each vertex v_i represents a process or a task in the program and each edge (v_i, v_j) indicates that process corresponding to vertex v_i communicates directly with the process corresponding to vertex v_j. Initially let us assume that each process is assigned to a different processor. For each program we really have a family of graphs, each corresponding to a problem size. The time to execute a parallel program depends on the problem size.

We had assumed that each process was assigned to a different processor. But on a real machine the number of processors available is bounded. The time to execute a parallel program depends on the number of available processors. So two

important factors in determining time complexity of a parallel program are problem size and number of processors available. Next we try to identify other factors. We have not yet accounted for the communication delays.

There are two important considerations in determining the time complexity. One is due to processing time and the other is due to the delay caused by communication. Processing time can be estimated by time to execute the program assuming communication delay is zero. Communication delays contribute significantly to the time complexity of a parallel program. We need to account for this. Communication delay can have two components. They are total message traffic and mean time to send one message.

We will model each of 1. processing time, 2. total message traffic, and 3. mean time to send one message as functions of number of processors and problem size.

Processing time is a function of problem size and number of processors. Let N indicate problem size and K indicate number of processors. PT indicates processing time

$PT = f_1(N, K)$

Y indicates total message traffic

$Y = f_2(N)$

Z indicates mean time to send one message

$Z = f_3(N, K)$

An Example

We have used mergesort algorithm for implementation. This algorithm was implemented on Intel iPSC (32 node) located at Argonne National Laboratories.

The algorithm is a typical divide-and-conquer algorithm. The list to be sorted is split into two sublists and each one is sorted using merge sort. Then the sorted lists are merged. Details of Intel iPSC computer are provided in Appendix A. Techniques for programming iPSC are given in Appendix B.

Using the estimated value of p we have tried to obtain an estimate of time complexity of our implementation of mergesort in terms of problem size and the number of processors (actually the dimension of the cube). We have fitted a quadratic response surface to the data. For the 3D plots of actual values and the predicted values, see Figures 5.1 and 5.2 respectively. The equation is:

$$Time(s,d) = 17.3 + .7d - 6.4\log(n) + .5d^2 - .5\log(n)d + .7\log^2(n)$$

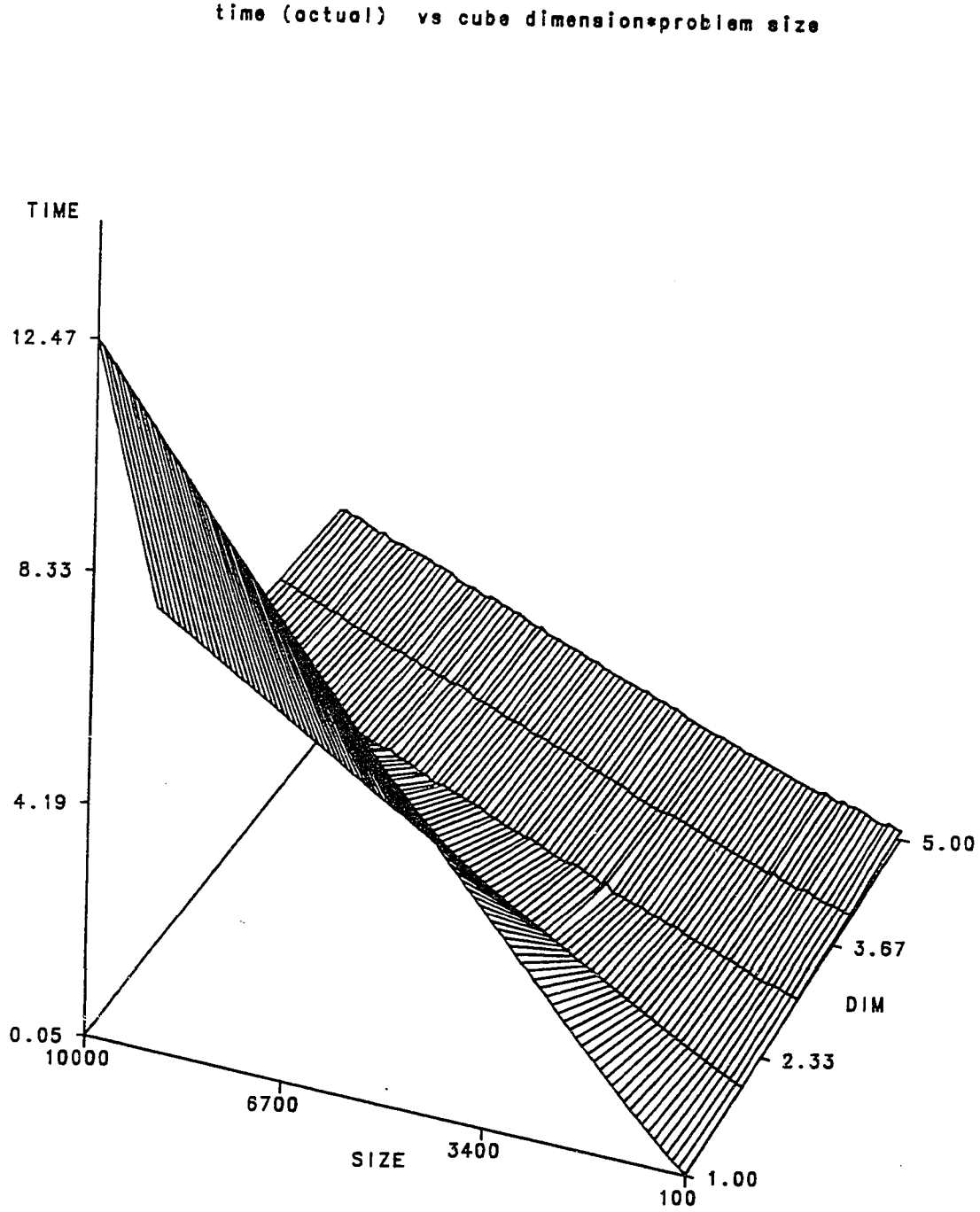

Figure 5.1: Plot of Time (actual) vs Cube Dimension*Problem Size

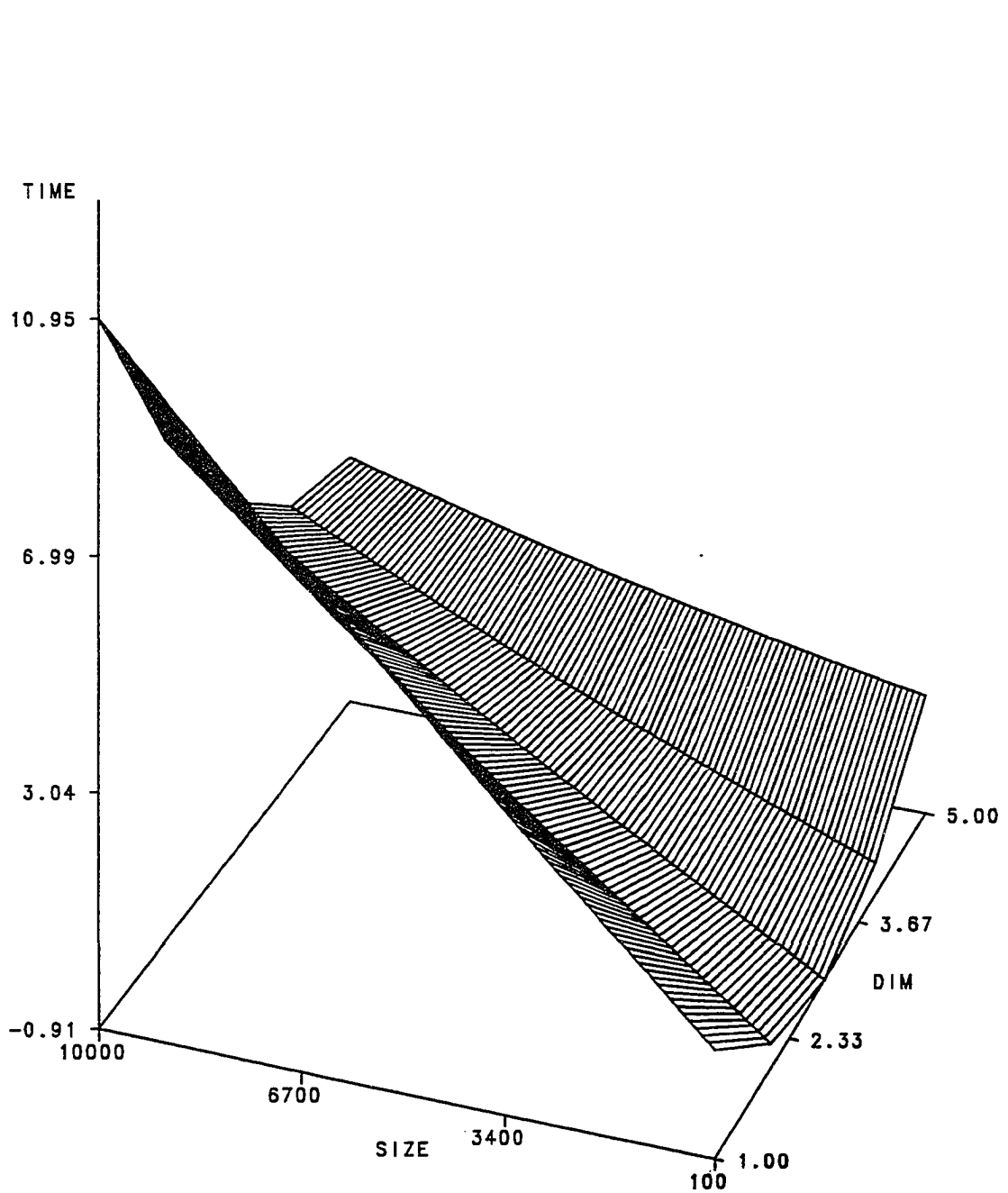

Figure 5.2: Plot of Time (predicted) vs Cube Dimension*Problem Size

CHAPTER 6. CONCLUSIONS

In this dissertation we addressed the question of how to make best use of parallel computers. While the problem of optimal utilization is very important, it is also very difficult. We have limited ourselves to a particular class of of machines called MIMD computers. There are many MIMD computers in the market. The range of problems that can be solved on MIMD computers is very large. We have restricted our investigation to a class of algorithms called divide-and-conquer algorithms. Divide-and-Conquer is a well known algorithm design technique. There is a large number of problems for which efficient divide-and-conquer algorithms exist. It would seem that divide-and-conquer is a natural candidate for parallel processing.

One of the major questions a user of a parallel computer would have to address is how to distribute the load among available number of processors. While a general answer does not exist, we felt that we needed to consider the underlying algorithm design paradigm. There are several such paradigms (i.e., divide-and-conquer, greedy, dynamic programming, etc.). In this work we looked in detail at the question of implementing divide-and-conquer algorithms on MIMD computers. We have to take into consideration the fact that the number of processors in a computer is fixed. In other words the computer can't expand to fit the process graph of a problem. We have developed techniques for implementing divide-and-

conquer algorithms on MIMD computers. For some divide-and-conquer algorithms we can bypass the process of mapping. This method is illustrated by implementing mergesort algorithm. But there are many divide-and-conquer algorithms that are not amenable to this method. Then we propose a generally applicable technique for implementing divide-and-conquer algorithms. The key to this techniques is the measure of similarity among processes that we defined. A notable aspect of this measure is that it considers not only the communication traffic between processes but also the contention for central processing unit (CPU) among processes that are assigned to the same CPU.

We have also addressed the question of obtaining a closed form expression to estimate the time complexity of a parallel program. Obtaining an expression analytically is extremely difficult. There are a number of factors that can't be measured accurately. Actually the problems are similar to those faced by economists and social scientists trying to model various phenomenon. In attacking the problem at hand my own background in statistics came very handy. In many complex situations where analytical methods seem very difficult, empirical methods provide a viable alternative. That is what we have done to estimate parallel time complexity. We have identified some key factors and obtained data. Then we used SAS to obtain a reasonable model. However, one has to be very cautious in interpreting the results outside the range of data used to obtain the expressions.

When we consider the big picture of parallel processing we have only attacked some small problems. There remains a great deal to be done. One can look at problems involved in implementing other algorithm design paradigms (i.e., dynamic programming, greedy method, etc.) on parallel machines. From our experience,

looking at implementations from the point of view of the underlying design technique is a very powerful approach. We tend to think that this approach should be usable for other paradigms. There is also scope for improvement is refining the measure of similarity among processes that we proposed. Perhaps more factors can be introduced in the computation of the measure. The framework we have used can easily allow this.

On the question of parallel time complexity there is a lot to be done. We have proposed using empirical methods. There is always scope for refining the methods. Perhaps studying distributional properties would be very interesting. This can also help in commenting about the reliability/extensibility of the expression obtained. Right now we need to be very cautious in using the expression for time complexity outside the range of data that is used for obtaining the expression. Some of the statistical tests can't be used because the underlying assumptions may not be valid. We have opened a whole new area for further research. It calls for close interaction of computer scientists and statisticians. We should not believe that empirical methods are a panacea. Analytical methods are very important. In fact we need sound analytical basis for using empirical methods. A lot needs to be done about computing concrete parallel time complexity.

BIBLIOGRAPHY

[1] Lee Adams. *High Performance Graphics in C:Animation and Simulation.* Windcrest Books, 1983.

[2] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison Wesley, Reading, MA, 1983.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[4] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, August 1986.

[5] Richard Anderson and Ernst Mayr. *Parallelism and Greedy Algorithms.* Technical Report STAN-CS-84-1003, Stanford University, Apr 1984.

[6] U. Bannerjee. *Direct parallelization of call statements-A review.* Technical Report 576, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Nov 1985.

[7] U. Bannerjee. *Speedup of ordinary programs.* Ph.D. Dissertation, Univ. of Illinois at Urbana-Champaign,Dept of Computer Science, Oct 1979.

[8] U. Bannerjee, S. C. Chen, and D. Kuck. Time and parallel processor bounds for fortran-like loops. *IEEE Transactions on Computers*, 660–670, sep 1979.

[9] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. In *Proceedings of International Conference on Parallel Processing*, 1984.

[10] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, 4:439–458, 1987.

[11] Dimitri P. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, AC-27(3), June 1982.

[12] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4), December 1980.

[13] G. Blelloch. *AFL-1:A programming language for massively concurrent computers.* Master's Thesis, Dept. of EE and Computer Science,MIT, Cambridge,Mass., June 1986.

[14] G. Blelloch. *Parallel prefix versus concurrent memory access.* Technical Report, Thinking Machines Corp., Cambridge,Mass., 1986.

[15] S. H. Bokhari. Partitioning problems in parallel, pipelined and distributed computing. *IEEE Transactions on Computers*, 37(1), 1988.

[16] George E. P. Box and Norman R. Draper. *Empirical Model-Building and Response Surfaces.* John Wiley & Sons, New York, 1987.

[17] J. C. Brown. Formulation and programming ofparallel computations: a unified approach. In *Proceedings of International Conference on Parallel Processing*, 1985.

[18] Gregory T. Byrd and Bruce A. Delagi. *Considerations for Multiprocessor Topologies.* Technical Report STAN-CS-87-1144, Stanford University, jan 1987.

[19] Nicholas Carriereo and David Gelernter. The s/net's linda kernal. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[20] Chatterjee and Hadi. *Sensitivity Analysis in Linear Regression.* John Wiley & Sons, New York, 1988.

[21] D. P. Christman. *Programming the Connection Machine.* Master's Thesis, Dept. of EE and Computer Science,MIT, Cambridge,Mass., January 1983.

[22] Norman Cliff. *Analysing Multivariate Data.* Harcourt Brace Jovanovich, 1987.

[23] Murray Cole. *Algorithmic Skeletons : Structural Management of Parallel Computation.* The MIT Press, Cambridge,Mass., 1989.

[24] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[25] Frank da cruz. *Kermit,A File Transfer Protocol.* Digital Press, 1987.

[26] John Dagpunar. *Principles of Random Variable Generation.* Oxford University Press, 1988.

[27] N. Deo and M. J. Quinn. Parallel graph algorithms. *ACM Computing Surveys*, 16(3), September 1984.

[28] Narsingh Deo. *Graph Theory With Applications to Engineering and Computer Science.* Prentice-Hall, Engelwood Cliffs, N.J., 1986.

[29] J. J. Dongarra and D. C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. Stat. Comput.*, 8(2), March 1987.

[30] J. G. Donnett and D. B. Skillicorn. Code partitioning by simulated annealing. In E. Chiricozzi and A. DAmico, editors, *Parallel Processing and Applications*, North-Holland, 1988.

[31] David Gelernter et al. Parallel programming in Linda. In *Proceeding of 1985 International Conference on Parallel Processing*, 1985.

[32] David Gelernter et al. *A symmetric Language.* Technical Report YALEU/DCS/TR-568, Department of Computer Science, Yale University, 1987.

[33] M. Flynn. Very high speed computing systems. In *Proceedings of IEEE*, pages 1901–1909, IEEE, 1966.

[34] Eli Gafni and Nicola Santoro, editors. *Distributed Algorithms on graphs.* Carleton University Press, 1986.

[35] C. W. Gear and R. G. Voigt. *Selected Papers from the Second Conference on Parallel Processing for Scientific Computing.* SIAM, 1987.

[36] David Gelernter. Domesticating parallelism. *IEEE Computer*, August 1986.

[37] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), 1985.

[38] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[39] Warren Gilchrist. *Statistical Modelling.* John Wiley, New York, 1984.

[40] Michael J. C. Gordan. *Programming Language Theory and its Implementation.* Prentice Hall, Engelwood Cliffs, N.J., 1988.

[41] Daniel H. Green and Donald E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkhauser, 1981.

[42] John L. Gustafson. The scaled-sized model:a revision of amdahl's law. In *Proceedings of International Conference on Supercomputing*, 1988.

[43] Bruce Hajek. A tutorial survey of theory and applications of simulated annealing. In *Proceedings of The 24th IEEE Conference on Decision and Control*, 1985.

[44] W. Händler. The impact of classification schemes on computer architectures. In *Proceedings of International Conference on Parallel Processing*, pages 7–15, IEEE, 1977.

[45] Wolfgang Händler. On classification schemes for computer systems in the post-von-neumann era. In *Lecture Notes in Computer Science 26*, pages 439–452, Springer, 1975.

[46] Gordan Harp. *Transputer Applications.* Pitman, 1989.

[47] Richard J. Harris. *A Primer of Multivariate Statistics.* Academic Press, New York, 1985.

[48] Paul Helman. *A common schema for dynamic programming and branch-and-bound type algorithms.* Technical Report CS86-4, Department of Computer Science, University of New Mexico, 1986.

[49] Paul Helman. *The principle of optimality in the design of efficient algorithms.* Technical Report CS85-10, Department of Computer Science, University of New Mexico, 1985.

[50] W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge,Mass., 1985.

[51] R. W. Hockney and C. R. Jesshope. *Parallel Computers.* Adam Hilger Ltd., 1981.

[52] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville,MD, 1984.

[53] E. Horowitz and A. Zorat. Divide-and-conquer for parallel processing. *IEEE Transactions on Computers*, C-32(6), June 1983.

[54] Hwang and Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, New York, 1986.

[55] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, Engelwood Cliffs, N.J., 1988.

[56] L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors. *The Characteristics of Parallel Algorithms.* MIT Press, Cambridge,Mass., 1987.

[57] Hong Jia-Wei. *Computation:Computability, Similarity and Duality.* John Wiley & Sons, Inc., New York, 1986.

[58] E. E. Johnson. Completing an MIMD multiprocessor taxonomy. *Computer Architecture News*, 16(3):95–101, June 1988.

[59] Richard A. Johnson and Dean W. Wichern. *Applied Multivariate Statistical Analysis.* Prentice Hall, Engelwood Cliffs, N.J., 1988.

[60] Maurice Kendall. *Multivariate Analysis.* Macmillan, 1980.

[61] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(2), February 1970.

[62] S. J. Kim and J. C. Browne. A general approach to mapping of parallel computations upon multiprocessor architectures. In *Proceedings of International Conference on Parallel Processing*, pages 1–8, 1988.

[63] Seungbeom Kim, Seungryoul Maeng, and Jung Wan Cho. A parallel execution model of logic program based on dependency relationship graph. In *Proceedings of International Conference on Parallel Processing*, 1986.

[64] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, May 1983.

[65] H. T. Kung and D. Stevenson. A software technique for reducing the routing time on a parallel computer with a fixed interconnection network. In David J. Kuck et al., editor, *High Speed Computer and Algorithm Organisation*, Academic Press, New York, 1977.

[66] J. B. Lasserre. A mixed forward-backward dynamic programming method using parallel computation. *Journal of Optimization Theory and Applications*, 45(1), January 1985.

[67] Chrles E. Leiserson and Bruce M. Maggs. Communication-efficient parallel graph algorithms. In *Proceedings of International Conference on Parallel Processing*, 1986.

[68] Clement H. C. Leung. *Quantitative Analysis of Computer Systems.* John Wiley, New York, 1988.

[69] Guo-Jie Li and Benjamin W. Wah. How good are parallel and ordered depth-first searches? In *Proceedings of International Conference on Parallel Processing*, 1986.

[70] Yow-Jian Lin and Vipin Kumar. A parallel execution scheme for exploiting and-parallelism of logic programs. In *Proceedings of International Conference on Parallel Processing*, 1986.

[71] H. Linhart and W. Zucchini. *Model Selection.* John Wiley, New York, 1986.

[72] A. C. Liu, Y. M. Zou, and D. M. Liou. Divide-and-conquer for a shortest path problem. In *Proceedings of International Conference on Parallel Processing*, 1987.

[73] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, 37(11), 1988.

[74] George S. Lueker. Some techniques for solving recurrences. *ACM Computing Surveys*, 12(4), December 1980.

[75] Mathematics and Computer Science Division. *Using the Intel iPSC.* Argonne National Laboratory, 1987.

[76] Ernst W. Mayr. *Well Structured Parallel Programs Are Not Easier to Schedule.* Technical Report STAN-CS-81-880, Stanford University, sep 1981.

[77] J. S. Milton and Jesse C. Arnold. *Probability and Statistics in the Engineering and Computing Sciences.* McGraw-Hill, New York, 1983.

[78] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. Convergence and finite-time behavior of simulated annealing. In *Proceedings of The 24th IEEE Conference on Decision and Control*, 1985.

[79] L. Mukkavilli and G. M. Prabhu. *Complexity of Parallel Divide-and-Conquer Algorithms.* Technical Report, Computer Science Department, Iowa State University, 1989. In Preparation.

[80] L. Mukkavilli and G. M. Prabhu. *On Implementing Divide-and-Conquer Algorithms for Multiprocessors.* Technical Report 89-1, Computer Science Department, Iowa State University, 1989.

[81] L. Mukkavilli, G. M. Prabhu, and C. T. Wright. Divide-and-conquer algorithms for multiprocessors. In *Proceedings of The Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, 1989.

[82] L. Mukkavilli, G. M. Prabhu, and C. T. Wright. Empirical modelling of parallel algorithm complexity. October 1989. Presented at First Great Lakes Computer Science Conference, Kalamazoo, MI.

[83] P. A. Nelson. *Parallel Programming Paradigms.* Ph.D. Dissertation, University of Washington, Seattle, 1987.

[84] R. H. J. M. Otten and L. P. P. P. van Ginneken. *The Annealing Algorithm.* Kluwer Academic Publishers, 1989.

[85] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of International Conference on Parallel Processing*, 1988.

[86] F. J. Peters. Tree machines and divide and conquer algorithms. In Wolfgang Händler, editor, *Lecture Notes in Computer Science 111*, Springer-Verlag, 1981.

[87] L. Pirktl. On the use of cluster analysis for partitioning and allocating computational objects in distributed computing systems. In James E. Gentle, editor, *Computer Science and Statistics: The Interface*, 1983.

[88] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[89] Terence W. Pratt. The pisces 2 parallel programming environment. In *Proceedings of International Conference on Parallel Processing*, 1987.

[90] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-hill Book Company, New York, 1987.

[91] Reuven Y. Rubinstein. *Monte Carlo Optimization, Simulation and Sensitivity of Queueing Networks.* John Wiley & Sons, New York, 1986.

[92] U. Schendel. *Introduction to Numerical Methods for Parallel Computers.* Ellis Horwood, 1984.

[93] J. E. Shore. Second thoughts on parallel processing. *Computers & Electrical Engineering*, 1:95–101, 1973.

[94] Margaret Simmons, Rebecca Koskela, and Ingrid Bucher. *Instrumentation for Future Parallel Computing Systems.* ACM Press, New York, 1989.

[95] J. B. Sinclair. Efficient computation of optimal assignment for distributed tasks. *Journal of Parallel and Distributed Computing*, 4:342–362, 1987.

[96] D. B. Skillicorn. *A Taxonomy for Computer Architectures.* Technical Report TR 88-217, Department of Computing and Information Science, Queen's University, 1988.

[97] Lawrence Snyder. A taxonomy of synchronous parallel machines. In *Proceedings of International Conference on Parallel Processing*, pages 281–285, IEEE, 1988.

[98] James R Thompson. *Empirical Model Building.* John Wiley, New York, 1989.

[99] Jeffrey D. Ullman. *Some Thoughts about Supercomputer Organisation.* Technical Report STAN-CS-83-987, Stanford University, oct 1983.

[100] M. A. Wong, S. E. Madnick, and J. M. Lattin. A graph-decomposition method based on the density contour clustering model. In *Computer Science and Statistics: The Interface*, 1982.

[101] Chuan-lin Wu and Tse-yun Feng. A software technique for enhancing performance of a distributed computing system. In *Proceedings of COMPSAC*, 1980.

[102] Yu-Wen and Dan I. Moldvan. Detection of and-parallelism in logic programming. In *Proceedings of International Conference on Parallel Processing*, 1986.

GLOSSARY

Many of the definitions are necessarily imprecise and subjective.

Cluster Analysis A statistical technique used to identify groups(or clusters) based on observations.

Decomposition Division of labor among various processing elements.

Distributed Memory Parallel Computer A parallel computer where there is no common memory accessible to all processing elements.

Divide-and-Conquer An algorithm design paradigm that involves dividing a problem into subproblems, then solving the subproblems and combining the solutions of subproblems to obtain solution to the original problem.

Empirical Methods An approach to model building where the basis is the data collected.

Hypercube An arrangement of processor elements where every processor has a unique address in the range 0 to $2^k - 1$. Processors whose addresses differ in exactly 1 bit are connected. k is called the order of the cube [90].

MIMD Multiple Instruction Multiple Data (Flynn's classification).

NP-Complete Informally, a problem that needs enormous amount of computation to solve.

Parallel Computer A computer capable of parallel processing.

Parallel Processing A kind of information processing that emphasizes the concurrent manipulation of data elements belonging to one or more processes solving a single problem [90]. Involves several processing elements.

Regression Analysis A statistical technique used to obtain a mathematical expression to describe the relationship between one variable (called dependent variable) and one or more variables (called independent variable(s)).

SAS Statistical Analysis System. A collection of several computer programs useful for statistical analysis.

Shared Memory Parallel Computer A parallel computer where all processing elements have access to a common memory.

SIMD Single Instruction Multiple Data (Flynn's classification).

Simulated Annealing An approximate optimization technique.

Supercomputer A general purpose computer capable of performing computations at a very high speed.

INDEX

APPENDIX A. HYPERCUBE COMPUTERS

Introduction

A hypercube is a multiprocessor system consisting of numerous processors communicating through an inter-processor connection network. Each processor has its own local memory module physically attached to it. Each processor and its associated memory constitutes a "node" of the hypercube.

The hypercube is a message-based Multiple Instruction Multiple Data (MIMD) machine. It has multiple independent processors, all running different processes which implement different segments of the problem. These processors communicate through explicit messages passed over the inter-processor connection network.

The topology of the hypercube is cubical. Because of its versatility, a number of other topologies like ring, mesh, etc., can be embedded in a hypercube. Some of the commercial versions of the hypercube that have been introduced in the market are intel's iPSC, NCUBE, AMETEK and Floating Point Systems T series.

Specifically, an n-dimensional hypercube consists of $N = 2^n$ nodes. Each node has a direct physical link with n other nodes. Each node has a unique address (id). These addresses range from 0 to N-1. If we encode these addresses using bit vectors then two nodes i and j are neighbors if the bit encoding of their addresses differ in exactly one bit. Fig A.1 shows The configuration of a 3-cube.

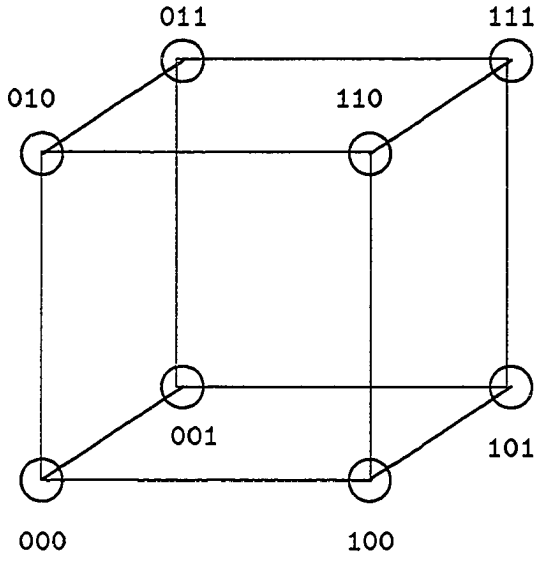

Figure A.1: The configuration of a 3-cube

The number of communication steps for a node i to communicate with another node j equals the number of bit positions in which the addresses of i and j differ. Since the address of every node in an n-cube can be encoded using bit vectors of length $log_2 N$, any two nodes can communicate using upto $log_2 N$ communication steps.

Basic Description of Intel iPSC

Intel iPSC is the first in a family of concurrent personal computers manufactured by Intel. It is designed to provide the research community with a reliable parallel computer and a basic programming enviornment. This could be used as a base to develop parallel programming techniques, program development tools and application programs [75].

The basic iPSC system consists of 2 main components:

1. Cube

   The cube consists of a set of microcomputers linked according to the hypercube topology using high speed communication channels. Each microcomputer constitutes a "node" of the hypercube and has its own numeric processing unit and local memory. All communication among nodes is in the form of messages passed which are queued at the destination nodes. This is handled at each node by the resident Node Operating System (NX) and dedicated communication co-processors.

2. Cube Manager

   The Cube Manager is a System 310AP microcomputer. This is connected to every node of the cube by an Ethernet communication channel. It provides a user interface to the cube. Specifically, it provides programming support, system management and diagnostics.

The basic iPSC System Structure is shown in Fig. A.2.

iPSC Hardware

The basic hardware for the various components of intel iPSC is described below

Node Hardware

- Each node is an independent computer based on Intel 80286 cpu and 80287 numeric processing unit.
- The local memory at each node consists of 512 Kbytes of NMOS dynamic RAM with byte parity.

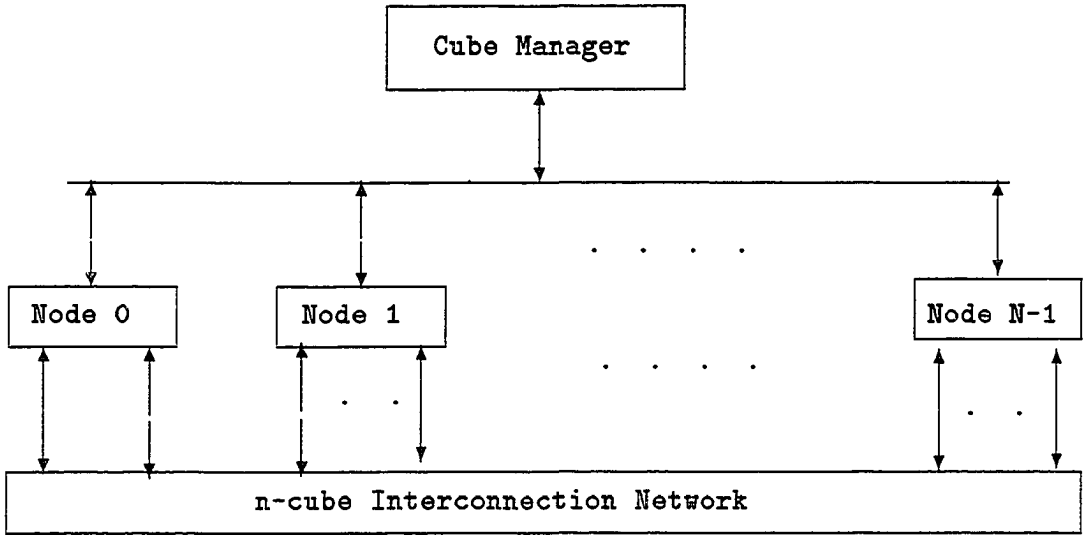

Figure A.2: iPSC System Structure

- Each node also has 64 Kbytes of PROM which contains an initialization program and a self-test monitor. This initializes the node at system power-up.

- There are 8 bidirectional communication channels connected to each node. These are managed by dedicated 82586 communication co-processors. Seven of these channels physically link the nodes together and are used for node-to-node communication. The eighth is a global ethernet channel which provides direct access to and from the cube manager.

Cube Manager Hardware

- The Cube manager is the system 310AP Multibus-based microcomputer which uses Intel's 80286 cpu and 80287 numeric processing unit. It contains a 140 megabyte Winchester disk, a 320 Kbyte floppy disk, a 45 megabyte cartridge

tape and 2 megabyte ECC RAM memory.

- There are 2 cube manager-to-node communication links. One is used for global communication with the cube nodes. The other is used for diagnostics. The former links the cube-manager to the global communications co-processor on each node. This is a standard ethernet link. The diagnostic link is a separate path for communicating with the nodes. In the event of a failure, this alternate path is used to determine if the fault lies within the node or in the global communications channel.

iPSC Software

The cube manager is the programming station which has the necessary software for program development, system management and diagnostics. On the other hand, the software on each node consists mainly of the resident Node Operating System (NX) and a Node Confidence Test (NCT). A brief description of each is given below.

Cube Software

1. Node Confidence Test (NCT) - This resides on the PROM and runs automatically when the system is switched on. It initializes each node by enabling node memory, communications controllers, I/O controller, interrupt controller and cpu.

2. Node Operating System (NX) - This is loaded onto each node after initialization. The basic functions of the NX are

    - *Inter Process Communication* - provides the users with a powerful set of communication routines. Sending and receiving of messages can be

synchronous or asynchronous.

- *Process Management* - processes on a node are executed in a time-shared manner.
- *Physical Memory Management* - provides memory for each process as well as for message buffering.
- *Protected Address Space* - protection of the Node Operating System is hardware enforced. Therefore a user process cannot corrupt the Node Operating System or any other process.

Cube Manager Software The cube manager software can be classified into 3 categories

1. Programming and Development Software -

   consists of the XENIX Operating System which is a derivative of the UNIX System 3. It also has all the standard XENIX tools (Microsoft's C compiler and other utilities) and additional tools developed at Intel.

2. Cube Manager Commands -

   These can be invoked from the terminal as commands. Some of the common functions performed by these routines are -

   - *Cube Access* - permits a user to gain exclusive access to the cube. There is also a command to release access to the cube. It should be noted that the hypercube is a single user system.
   - *Load Cube* - permits the user to load the nodes with the Node Operating System or the application processes.

- *Maintain a System Logfile* - keeps a log of system events. Each entry is date and time stamped so the sequence of events is preserved.

3. Diagnostics -

   Diagnostic Software is mainly of 2 types - confidence tests and diagnostic tests. The confidence tests are used to check the integrity of the system prior to usage of the cube. The diagnostic tests on the other hand are used to isolate the faults at the board or system module level.

Programming Concepts for Intel iPSC

A parallel processing application program on the iPSC is implemented as a set of "node" processes and "cube-manager" (host) processes. Typically, a parallel application first requires distributing the input data from the cube manager to the nodes of the cube to set-up the initial data configuration. After receiving the data, each node process does its computation, communicating with other nodes for intermediate results if required and the final results are then communicated back to the host process.

The strategy for programming on the iPSC is -

- Decompose the problem into several independent parts. This decomposition process is constrained by the data dependencies of the computation involved in the problem. Each of the decomposed parts would correspond to a process. These processes could be the same program operating on different sets of data or unique programs. The structure of processes of a problem and their interaction can be represented as a process graph in which the nodes would

represent processes and an edge between two nodes signifies communication between the two processes.

- The next step is to place these parts onto the various nodes. In order to optimize the efficiency of the application, the processors need to be placed in such a manner as to minimize the communication steps required. Also, the processes need to be so designed and placed such that the entire workload is evenly distributed among the nodes.

Introduction to cube manager and node libraries

Communication among processes is crucial for any parallel processing application and this has to be explicitly handled by the programmer using routines provided by iPSC [75].

Before a host or node process can communicate with another process it needs to open a *communication channel.* The copen routine is used to create a communication channel and may be called within a node or host program. The cclose routine on the other hand is used at the end, to destroy any specified communication channel created previously.

The cube manager operating under XENIX, treats a channel as a device. A host process running on the cube manager can execute one device-related routine at a time. Consequently, a host process can send/receive only one message at a time. The nodes on the other hand have their own operating system which can support multiple channels. Node processes can therefore perform concurrent sends and receives if they have multiple channels open. On any particular channel the messages are sent sequentially.

Once a communication channel has been created by a process, it can start sending and receiving messages over that channel. Due to the different operating systems on the cube manager and the nodes of the cube, the procedures for sending and receiving messages are also different for the cube manager and the nodes.

The Node Operating System supports message passing both in synchronous and asynchronous mode. **send** and **recv** are non-blocking asynchronous routines. They initiate the transmission (receipt) of a message. When issued they return to the calling process as soon as the Node Operating System records the request. It is therefore the user's responsibility to check whether the operation has been completed using **status** routine. **sendw** and **recvw** routines on the other hand, are blocking, synchronous procedures. They return to the calling process only after the Node Operating System has actually finished the operation. Completion of a **sendw** operation does not imply the receipt of the message by the destination process, but merely that it has been sent.

sendmsg and **recvmsg** are the procedures for message passing on the cube manager. These are synchronous routines and very similar to **sendw** and **recvw** .

type is one of the parameters in send and receive routines. This is useful from a programmer's point of view as its value is user-definable (0 - 32767) and can be used by the programmer to identify different classes of messages. It should not be confused with variable types as supported by High Level Languages.

Node processes qualify message reception on the basis of **type**. A node process accepts a received message from the queue only if its **type** matches the request. Therefore, different values of **type** should be used to prevent messages from different processes or nodes from getting mixed up at the receiving node. On the cube

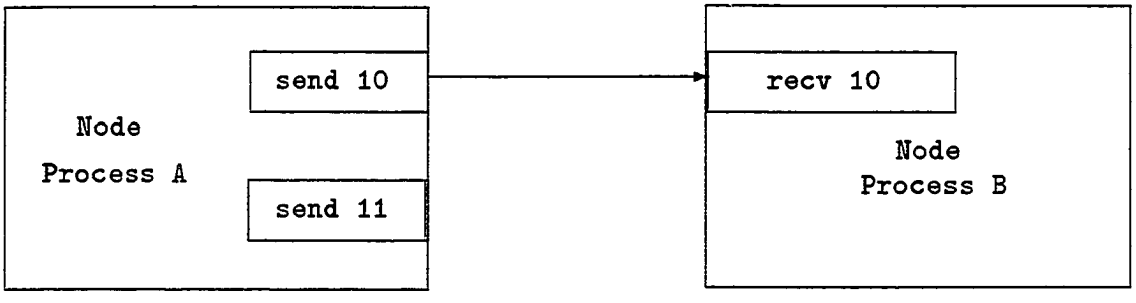

Figure A.3: Only message type 10 can be received by node process B

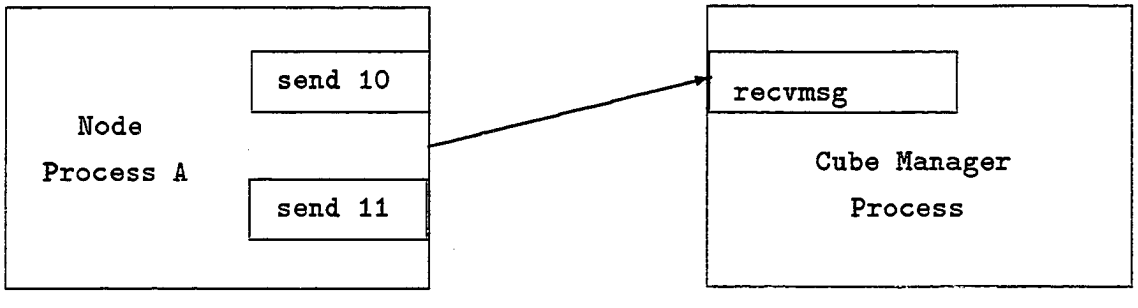

Figure A.4: The first message to arrive at the Cube Manager will be received

manager however, the **recvmsg** procedure call can be fulfilled by a received message of any **type**. The **type** parameter in the **recvmsg** call is returned as a result of the call. Figs. A.3 and A.4 illustrate the use of **type**.

The Execution Environment

All application programs run as a set of processes. They reside in the available local memory at each node and the cube manager. The processes communicate through communication channels. Buffers are used at each node to store messages that are either destined for that node or are transitory in nature. The 3 important

aspects of the execution enviornment are discussed below [75].

1. **Processes -**

   An application program would consist of a set of routines or processes. A copy of the node programs reside in each node's memory. These execute concurrently on the nodes and the cube manager.

   Node processes differ from host processes as they are incapable of I/O. They rely on the cube manager process for I/O.

   iPSC allows multiple processes at each node. The 5 dimensional iPSC has an upper limit of 20 processes per node. The actual number of processes that can run on a node is determined by the following -

   - The amount of local memory available at each node. Even though each node has 512 Kbytes of memory associated with it, the NX (Node Operating System) and message buffers occupy roughly 200 Kbytes of it, leaving 312 Kbytes of available memory for the node processes.
   - The number of buffers actually allocated. The number of buffers to be allocated is specified when loading NX with the load command. The allowed minimum and maximum for a 32-node iPSC are 25 and 300 respectively and the default is 100 buffers per node.
   - The size of the individual processes.
   - The use of dynamic storage allocation (using "malloc" in C). NX allocates 64 Kbytes of memory at a time to facilitate the supply of memory to "malloc" calls. If the size of avilable memory is less than 64 Kbytes then all of it is allocated.

Processes on the iPSC are referred to by a *process id* (pid) and the *node id* (nid). Id's are assigned to a process when it is loaded onto the nodes of the cube. On the other hand the host process id's are assigned by XENIX when the host process is executed. For an n-dimensional cube the node id's range from $2^n - 1$. The cube manager has a fixed id of -32768 (0x8000).

Processes pass messages among each other to communicate intermediate results of their computation and to co-ordinate activities. The destination pid and nid are specified as parameters in the **send** routine call. The actual routing of messages is transparent to the user and is done automatically by NX using the shortest paths. The messages sent are queued at the destination. Usually the sender specifies a value for **type** in the call. The destination node checks for a match of the received **type** value with the specified value of **type** in the **recv/recvw** routine call. The message is accepted only if a match occurs.

All messages are passed by "*value*" and not "*by reference* " as there is no concept of shared memory in the intel iPSC. This also applies to processes communicating within a node as they are treated exactly the same as processes communicating across nodes.

2. **Buffers -**

NX uses System Buffers to store messages. The number of buffers to be allocated needs to be specified during system startup using **load** command. The space occupied by the buffers is not available for the user processes.

Each buffer is 1044 bytes in length. A 5-dimensional intel iPSC has an *allowed* minimum and maximum of 25 and 300 buffers respectively. This range may not coincide with the *effective* minimum and maximum of any particular application. For example, an application may require at least 75 buffers per node to execute even though the allowed minimum is 25. On the other hand by allocating a large number of buffers (say 300), the size of available memory may be reduced to such an extent that it may not be possible to load the user processes onto the node itself. Therefore, the optimal number of buffers needs to be determined for any application. This may be done by doing a worst case analysis (in terms of the messages passed) of the program or more simply by using a hit and trial approach. This approach is to do nothing unless the program hangs and if so, increase the number of buffers allocated.

3. **Channels -**

A channel is a 64 byte block of memory that holds information about a message sent or received. The information in a channel consists of the *source nid and pid, destination nid and pid, message length* and *message location* .

A channel is created by the **copen** routine. The same channel can be reused for consecutive send's/receive's. But, if a process needs to send/receive multiple messages simultaneously then it needs to open a channel for each of the simultaneous send's/receive's.

At this point, it should be noted that it is important to separate consecutive runs of an application by flushing all outstanding messages as they may interfere with the next run. The cube manager command **loadkill** may be used

between runs for this purpose.

NX flushes all messages in a node, regardless of pid when the cube manager uses the lkill routine and specifies all processes (-1) on that node. The cube manager routines lwait or lwaitall also flush all messages destined for a specific process at the completion of that process.

APPENDIX B. PROGRAMMING INTEL iPSC

The source code for a parallel application has to be compiled to obtain the executable version of the program. Since most parallel application programs consist of node and host programs , both need to be compiled separately. The executable files of the node programs can then be loaded onto the cube and executed. This chapter will highlight the steps involved in preparing the source code for execution and the command sequence used to finally run the parallel program [75]. All this will be illustrated by means of a sample application program. The final section provides some helpful tips on debugging parallel programs on the Intel iPSC.

Development Steps

Let us assume that the cube manager (host) and node programs reside in the files "host.c" and "node.c" respectively. We implicitly make the assumption that the node program is the same for each node even though different nodes may do different computations based on their node's id. The invocation sequences for compiling the host and node programs are different and hence they have been treated separately.

Developing cube manager processes

The development of the cube manager process consists of compiling all the host programs separately. If the host program uses math libraries then the user has an option of using either Intel or Microsoft math libraries.

The invocation sequence for compiling "host.c" is given below. This assumes that there is a single host program.

```
cc -Alfu -o host host.c /usr/ipsc/lib/Llibcel.a
   /usr/intel/lib/cel287.a /usr/ipsc/lib/chost.a
```

where:

- -Alfu : creates a file to conform to Intel's "large" model. However to use data objects larger than 64 Kbytes, the -Alhu option has to be specified in order to use the "huge" model.
- -o host : Names the output file "host".
- host.c : Name of the file to be compiled.
- /usr/ipsc/lib/Llibcel.a
  /usr/intel/lib/cel287.a : Intel math libraries. The Microsoft math libraries can be used instead of the Intel math libraries by specifing -lm instead of the Intel library names. This option links in Microsoft's math library "Llibm.a".

Developing Node Processes

The development of a node process consists of 2 steps - compilation and binding. Again, in case there is more than one node program then each will have to be compiled separately.

The invocation sequence for

1. Compilation -

```
cc -Alfu -K -O -c node.c
```

where :

- -Alfu : Creates a file to conform to the "large" model. The other option is to use the "huge" model (-Alhu).
- -K : Does not include stack probes in the output module.
- -O : Optimizes the generated code.
- -c : Does not invoke "ld". Stops after producing the ".o" file, without binding.
- node.c : Source code file name.

2. Binding -

```
ld -Ml -o node /lib/Lseg.o /usr/ipsc/lib/Lcrtn0.o node.o \
    /usr/ipsc/lib/Llibcnode.a /usr/ipsc/lib/Llibmnode.a
```

where :

- -Ml : Creates a "large" model C application.
- -o node : Names the executable file "node".
- /lib/Lseg.o : Defines "large" model segments.
- /usr/ipsc/lib/Lcrtn0.o : C runtime start-up library.
- node.o : Name of file to be linked. All the compiled modules to be linked must be listed here.
- /usr/ipsc/lib/Llibcnode.a : C library containing versions of malloc, free, calloc, realloc and sbrk which get memory directly from the NX.
- /usr/ipsc/lib/Llibmnode.a :
  The Intel math libraries "Llibcel.a" and "cel287.a" may be substituted for the Microsoft math library "Llibmnode.a".

The steps involved in developing a parallel application on the Intel iPSC can be greatly simplified by the use of makefiles. These can be used to develop both the the host and node programs. A copy of this makefile exists in "/usr/ipsc/examples/nx_c/ring" in a file called "makefile". This makefile can be tailored to suit any particular application by substituting the host program name for "host" and the name of the node program for "node".

A Sample Application

The parallel application program being considered in this section is for broadcasting a message (character string) from the cube manager to all nodes of the hypercube. The host program prompts the user for the message that is to be

broadcast. The inputted string is then sent to node 0, which acts as the *source* and broadcasts the received message to the other nodes.

This broadcasting proceeds along the branches of a Binomial Spanning Tree embedded in the cube and rooted at node 0. This will be discussed in greater detail in Chapter 5. For the moment, based on the embedding of the above tree, we can classify the nodes of the cube as *leaf* nodes and *non-leaf* nodes. The *leaf* nodes are those that have a 1 as the highest order bit in the binary encoding of their node id's. The rest constitute the *non-leaf* nodes. A *non-leaf* node i receives the message from its parent which is uptree and then sends it to all its *children* . The node id's of the children are determined by complementing the leading zero's (one at a time) in the binary encoding of i. Therefore all the children of any node i are neighbors of i. The broadcast operation is over once all the *leaf* nodes receive the message. Every node, *leaf* and *non-leaf*, acknowledges the receipt of the message by sending the message back to the cube manager. The communication between nodes is shown in Fig. B.1.

It should be noted that even though the *leaf* and *non-leaf* nodes perform different set of actions, the node program as listed below is the same for all the nodes. Typically, the node program for any parallel application is organized such that the different nodes perform distinct pieces of the same program. This approach simplifies the development process.

The cube manager process can be compiled by using the invocation sequence -

```
% cc -Alfu -o host host.c /usr/ipsc/lib/chost.a
```

or

```
% make host
```

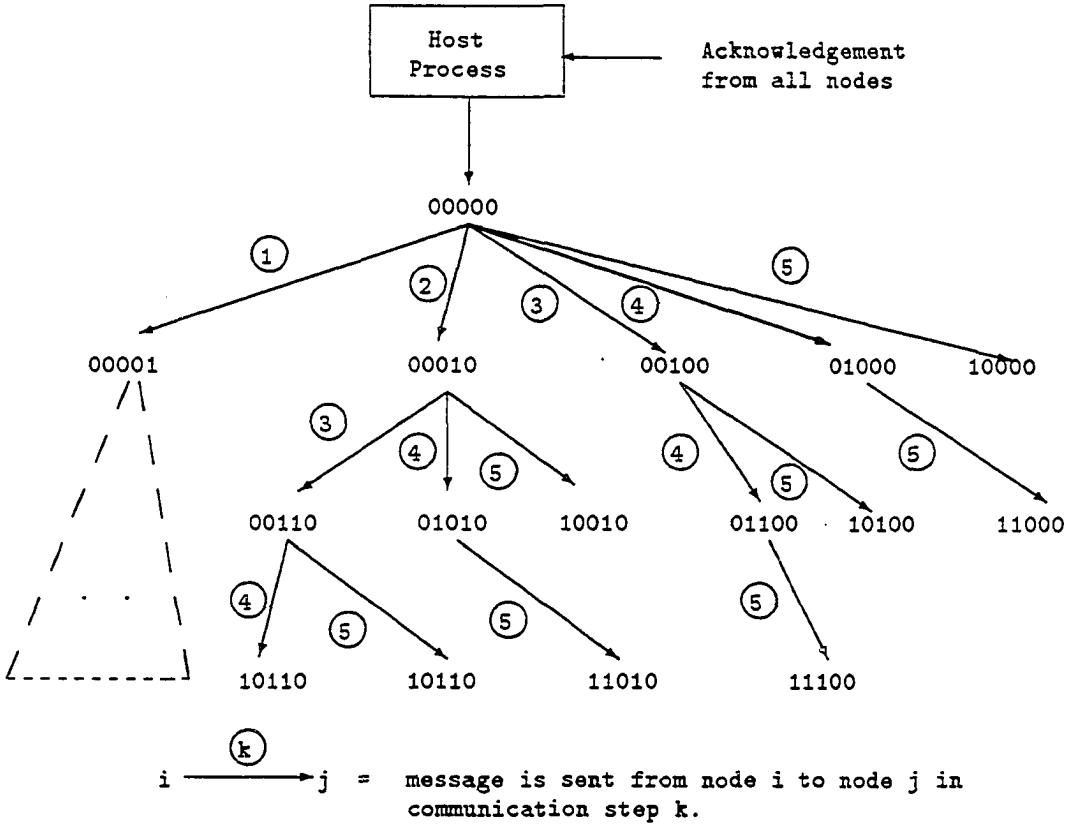

Figure B.1: The Structure of Communication in the Broadcast Application

This produces the executable file "host" on the cube manager. The node program may be compiled and bound by using the invocation sequence -

```
% cc -Alfu -K -O -c node.c
% ld -Ml -o node /lib/Lseg.o /usr/ipsc/lib/Lcrtn0.o \
      node.o /usr/ipsc/lib/Llibnode.a
```

or

```
% make node
```

This produces the executable file "node" on the cube manager. This can now be loaded into the cube. It is simpler to generate both processes using the "make" facility.

```
% make both
```

These processes can be run by using the following command sequence.

```
% getcube
```

Obtains exclusive access to the cube. In case the cube is currently being accessed, an appropriate message is displayed and the user has to try again.

```
% cubelog -l logf
```

Defines a log file "logf" (user specified name) in the current directory. Transfers logging from the system logfile "LOGFILE" in "/usr/ipsc/log" to "logf" .

```
% tail -f logf &
```

Used to display log file entries on the screen if desired.

```
% load -c
```

loads the NX into the cube. This is executed only once before the first run of the application. It need not be used between runs.

% **host**

Run the application. The host process loads the node processes using "load" routine. It is also possible to load the node processes by using the "LOAD" command.

Debugging

The simplest way of debugging C programs on the Intel iPSC is by having print statements as check points to trace the control flow in the source code. The nodes of the cube are incapable of I/O so two alternative ways are used to print the debug messages.

- Using syslog routine.
- Packing the debug messages and sending them to the cube manager. The host process can receive these messages and print them out using standard XENIX I/O routines.

The syntax of syslog routine call is given below.

```
int mypid ;
char message[50] ;

syslog(mypid , message) ;
```

Textual messages as well as numeric data can be written into the message buffer message[ ] by using sprintf . A call to syslog results in an entry being made in the user specified log file (the default is /usr/ipsc/log/LOGFILE). These entries are time stamped and also include the id of the node (i.e., mypid) where it was invoked.

The other alternative is to explicitly package debug messages and send them to the cube manager for display on the terminal. One way of implementing this is by having a separate host process which waits for debug messages from the nodes and prints them out. In this way debug messages can be handled separately from the normal messages that are passed from the nodes to the cube manager.

COLOPHON

This dissertation was typeset using ISUTHESIS style with LaTeXDocument Preparation System. ISUTHESIS was developed by Iowa State University Computation Center, LaTeXwas developed by Leslie Lamport. LaTeXis a special version of TeX, a computer typesetting system developed by Donald Knuth at Stanford University. The document was printed on XEROX 450 printer. The main body of the text is set in Computer Modern Roman. This typeface was developed by Donald Knuth.

The index was prepared using a program called *MakeIndex* written by Pehong Chen at University of California, Berkley. The graphs were made using SAS/GRAPH package and plotted on XEROX 450 printer.