PARALLEL MATRIX INVERSION ON A SUBCUBE-GRID

Dissertation submitted by

Sudhir Kumar Gangwar

In partial fulfilment of the requirements for the degree of

Master of Technology
in
Computer Science And Technology

1520

92p + bib.

School of Computer and Systems Sciences

Jawaharlal Nehru University

New Delhi

December 1993

CERTIFICATE

This is to certify that the dissertation entitled "Parallel Matrix Inversion on a subcube grid", being submitted by me to school of Computer and system sciences, Jawaharlal Nehru University, New Delhi in the partial fulfilment of the requirements for the award of the degree of Master of Technology, is a record of original work done by me under the supervision of Dr. C.P. Katti, Associate Professor, School of Computer and systems sciences, Jawaharlal Nehru University during the year 1993, Monsoon Semester.

The results reported in this dissertation have not been submitted in part or full to any other University or Institute for the award of any degree or diploma, etc.

Sudhir Kumar Gangwar
30/12/93

3-1-94

Prof.K.K.BHARADWAJ
Dean,
School of Computer
and system sciences,
J.N.U., New Delhi.

DR.C.P. Katti
Associate Professor
School of Computer
and System Sciences,
J.N.U., New Delhi.

ACKNOWLEDGEMENT

I express my sincere thanks to Dr. C.P. Katti, Associate Professor, School of computer and system sciences, Jawaharlal Nehru University, New Delhi for suggesting such a brilliant topic. I am indebted to him for his personal involvement with my work and his immense and eloquent guidance and encouragement which has been indispensable in bringing about a successful completion of the dissertation.

I am also grateful to my friends especially to Mr.Anil Kumar Tripathi for their valuable suggestions, all faculty and staff members who helped me in every way possible.

SKumar.
30.12.93
Sudhir Kumar Gangwar

CONTENTS

Page No.

CHAPTER I INTRODUCTION

1.1 Parallel Proceeding

1.2 Parallel Distributed Systems

1.2.1 Evolution & Area of Applications

1.2.2 Parallel & Distributed Systems : Difference

1.2.3 Parameters of Classification

1.2.4 Shared Memory & Distributed Memory Systems

1.3 Matrix Computations

1.4 Problems & Computer Implementation

1.5 Characteristics of Parallel Algorithms

1.6 Relevance of Project

CHAPTER 2 ARCHITECTURE & NETWORK TOPOLOGIES

2.1 Architectures

2.2 Network topologies

2.3 Course gran & Fine Grain Systems

2.4 Choice of System & Architecture

2.5 Communication Aspects

CHAPTER 3 DEVELOPMENT OF PARALLEL ALGORITHM FROM SEQUETIAL ALGORITHM

3.1 How to Achieve Parallelism

3.2 Parallelization

CHAPTER 4 COMPLEXITY ANALYSIS

4.1 Complexity of Algorithms

4.2 Bounds on Complexity

4.3 Conditioning of the Problem

4.4 Speed up and efficiency

4.5.1 Numerical Accuracy

4.5.2 Efficiency
4.5.3 Run Time & Communication Overhead

4.6 Calculation of COmplexities.

CHAPTER 5 IMPLEMENTATION AND LOAD BALANCING

5.1 Implementation

5.2 Load Balancing

CHAPTER 6 CONCLUSION

CHAPTER 1

INTRODUCTION

INTRODUCTION

1.1 PARALLEL PROCESSING

Parallel Processing is so much a feature of the universe that we are not normally concerned with it at all. However it is worthwhile to reflect on the contrast between the concurrent nature of the world. and the sequential nature of the digital computer. Since the main purpose of the computer is to model the world, there would seem to be a serious mismatch.

To adequantly model the concurrency of the real world, it would be preferable to have many processors all working at the same time on the same program. There are also huge potential performance benefits to be derived from such parallel processing. For regardless of how far electronic engineers can push the speed of an individual processor, 10 of each them running concurrently will still execute 10 times as many instructions in a second.

The Classical computer is a single-processor system, while the brain has a very large number of processing units that operate at the same time. Modern semiconductor technology has reached the level at which building a computer with thousands of process is possible. These systems are called massively parallel computers.

The two major parts of the classical Von Neumann computer are the central processing unit and the memory. Almost all of the hardware if in the memory, and only a small percentage is in the central processing unit. The central processing unit either reads or writes into one memory location at one time. This means that the central processing unit is busy all the time, but that most of the memory is idle. a large percentage of the computer hardware, say 95 percent, sits idle all the time. an obvious way for better utilization of the hardware is an architecture based on a large number of parallel processing units.

1.1.1 HARDWARE AND ECONOMIC REASONS FOR PARALLEL PROCESSING

Another example of the effect of changing technology on the desirability or feasibility of prallel organizations is afforded by recent work on semiconductor memories. the use of small distributed memories in now feasible since LSI memory costs tend to be linear with size. Hence, it is no longer economically necessary to design around a large central memory. The total internal memory capacity can be distributed over a number of smaller memory modules which can be accessed in parallel.

1.2 PARALLEL AND DISTRIBUTED ARCHITECTURES

Parallel and distributed computation is currently an area

of intense research activity, motivated by a variety of factors. There has always been a need for the solution of very large computational problems,but it is only recently that technological advances have raised the possibility of massively parallel computation and have made the solution of such problems possible.Furthermore,the availability of powerful parallel computers is generating interest in new types of problems that were not addressed in the past. Accordingly,the development of parallel and distributed algorithms is guided by this interplay between old and new computational needs on the one hand,and technological progress on the other.

1.2.1 EVALUATION AND AREA OF APPLICATIONS

1. The original needs for fast computation have been in a number of contexts involving partial differential equations (PDEs),such as computational fluid dynamics and weather prediction,as well as in image processing , etc. In these applications, there is a large number of numerical computations to be performed.The desire to solve more and more complex problems has always been running ahead of the capabilities of the time,and has complex problems has always been running ahead of the capabilities of the time ,and has provided a driving force for the development of faster,and possibly parallel,computing machines.The above mentioned types of problems can be easily decomposed along a spatial

dimension,and have therefore has prime candidates for parallelization ,with a different computational unit (processor) assigned the task of manipulating the variables associated with a small region in space. Furthermore , in such problems ,interactions between variables are local in nature,thus leading to the design of parallel computers consisting of a number of processors with nearest neighbour connections.

2. Recently, there has been increased interest in other types of large scale computation. Some examples are the analysis,simulation,and optimization of large scal interconnected system , queneing systems being a noteworthy representative .Other examples relate to the solution of general systems of equations, mathematical programming ,and optimization problems.A common property of such problems,as they arise composition tend to be fewer and more complex than those obtained in the context of is lost. Accordingly , one is led to use fewer and more powerful processors,coordinated through a more complex control mechanism. In both of the above classes of applications,the main concerns are cost and speed:the hardware should not be prohibitively expensive,and the computation should terminate within an amount of time that is acceptable for the particular application.

3. A third area of application of parallel,or rather distributed, computation is in information

acquisition, information exptaction , and control , within geographically distributed systems.

1.2.2 PARALLEL AND DISTRIBUTED SYSTEMS: DIFFERENCE

An important distinction is between parallel and distributed computing systems. Roughly speaking,parallel computing systems consist of several processors that are located within a small distance of each other.Their main purpose is to execute jointly a computational task and they have been designed with such a purpose in mind; communication between processors is reliable and predictable. Distributing systems are different in a number of ways. Processors may be far apart, and interprocessor communication is more problematic.

1.2.3 PARAMETERS OF CLASSIFICATION

There are several parameters that can be used to describe or classify a parallel computer and we refer to these briefly.

(a) **Type and number of processors:** There are parallel computing systems with thousands of processors .Such systems are called massively parallel,and hold the greatest promise for significantly extending the range of practically solvable computational problems. A diametrically opposite option is coarse-grained parallelism, in which there is a small number of

processors,say of the order of 10. In this case,each processor is usually fairly powerful, and the processors are loosely coupled, so that each processor may be performing a differeet type of task at any given time.

(b) **Presence or absence of a global control mechanism :** Parallel computers almost always have some central locus of control.A related popular classification along these lines distinguishes between SIMD (Single Multiple Data) and MIMD (Multiple Instruction Multiple Data)parallel computers,referring to the ability of different processors to execute different instructions at any given point in time.

(c) **Synchronous vs. asynchronous operation** : The distinction here refers to the presence or absence of a common global clock used to synchronize the operation of the different processors . Such synchronization is present is SIMDmachines, by definitaion. Synchronous operation has some desirable properties: the behavior of the processors is much easier to control and algorithm design is considerably simplified. On the other hand,it may require some undesirable overhead and , in some contexts,synchronization may be just impossible.For example,It is quite hard to synchronize a data communication network.

(d) **Processor interconnections** : A significant aspect of parallel computers is the mechanism by which processors exchange information. Generally speaking, there are two extreme alternatives known as shared memory and message-

passing architectures,and a variety of hybrid designs lying in between . The first alternative uses a global shared memory that can be accessed by all processors. A processor can communicate to another bywriting into the global memory , and then having then second processor read that same location in the memory.

In the second major approach,there is no shared memory,but rather each processor has it own local memory.Processors communicate through an interconnection network consisting of direct communication links joining certain pairs of processors,as shown in Fig. 2.

1.2.4 SHARED MEMORY & DISTRIBUTED MEMORY SYSTEMS

The characteristics of parallel algorithms are intimately interwined with the characteristics of the problem to be solved and the computer architecture on which the algorithm will be implemented. We use the term "architecture" to include the programming environment and operating system support, as well as machine hardware. Probably the most significant characteristic of parallel architectures is the organization of memory, specifically whether each processor has acess only to its own private local memory, or memory is globally shared among all processors (of course, hybrid systems may have both types of memory) . In a distributed-memory system, problem data (matrices, Vectors, etc.) must be partitioned and distributed across the individual processor memories, while in a shared-memory system all problem data reside in a common global memory where they are accessible by all processors. This means that standard data structures used on traditional serial machines for matrices and vectors usually carry over to shared-memory multiprocessors, whereas distributed-memory systems often require new distributed data structures, which can adversely affect the performance of parallel algorithms. Distributed data structures for sparse matrices, for example, tend to incur a penalty, both in storage overhead and efficiency of access, relative to their serial or shared-mêmory counterparts due to the loss of context that results from scattering the data across local processor memories..

In a distributed-memory multiprocessor, access to non-local data is provided by passing messages among the processors through an interconnection network. Common topological structures for such networks include rings, meshes, trees, and hypercubes. Analogously, the processors in a shared-memory system are connected to the common global memory by a bus or a switch, such as a crossbar switch or a multistage switching network. Thus, 2interprocessor communication bandwith is a key perofmance parameter for distributed-memory systems. While memory bandwidth is a correspondingly important parameter for shared-memory systems for a discussion of multiprocessor networks and architectures.

LOAD BALANCING

Another critical difference between shared-memory and local- memory systems is the manner in which the computational work load is distributed across the processors. Good efficiency requires that the work load be reasonably well balanced among the processors. In a shared memory system, a good load balance is fairly easily maintained by dynamically assigning work from a common pool of tasks. Such a scheme tends naturally and effectively to accommodate varying numbers of processors and relatively heterogeneous tasks, since processors are assigned new tasks as they become available. Such a common pool of tasks is not easily implemented without access to shared memory, however, so that distributed memory algorithms usually depend on a static assignment of work to

processors that is determined in advance of the computation. This places a significantly greater burden on the algorithm designer to ensure a good load balance.

1.3 MATRIX COMPUTATIONS

The fundamental importance of linear algebra problems in science and engineering has placed algorithms for matrix computations in the forefront of research on parallel algorithms.

Matrix computations are among the cornerstones of scientific computing. Linear algebra problems, including systems of linear equations, linear least squares problems, and algebraic eigenvalue problems, are fundamental to the computational solution of differential equations, optimization problems, and the analysis of various discrete structures. The effective use of parallel computer architectures in scientific computations is therefore critically dependent on exploiting parallelism in matrix computations. Matrix algorithms have been in the vanguard of algorithm development on multiprocessors not only because they are building blocks on which many other scientific computations are based but also because they serve as realistic prototypes that present many of the fundamental challenges of parallel computation in a pure form. Thus, the development of parallel algorithms for matrix computations has received strong emphasis from researchers in parallel computing, both as a tool and as a

paradigm for scientific computing in general on parallel architectures.

1.4 PROBLEMS AND COMPUTER IMPLEMENTATIONS

The most common computational problems in linear algebra are the solution of systems of linear algebraic equations and algebraic eigenvalue problems for various types of matrices. Significant problem characteristics that affect any computer implementation include whether the matrix is square or rectangular, symmetric or nonsymmetric, dense or sparse, explicitly represented by its entries or implicitly represented by its action on vectors. Such properties determine what algorithm , or family of algorithm is appropriate for solving the problem in any computing environment, whether serial or parallel, but may have more dramatic impact in the latter case. For example, the necessity of row or column interchanges for numerical stability can be a much more serious complication in a parallel environment than in a serial one. Another important problem characteristic in any computing environment is the size of the matrix, which determines the computational resources (processor time, storage) that will be required, as well as what classes of algorithms and data structures may be appropriate.

1.5 CHARACTARISTICS OF PARALLEL ALGORITHMS

1.5.1 CONCURRENCY, COMMUNICATIONS & GRANULARITY

The salient characteristics of parallel algorithms for matrix computations fall under the two main headings of "concurrency" and "communication". By concurrency we mean the overlapped or simultaneous execution of computations on multiple processors. The degree of concurrency is determined by the manner in which the overall computation is broken up into subtasks or subunits of computation, specifically their size distribution, scheduling, and mutual data dependence. In most computations, parallelism can potentially be exploited at any of a number of levels, depending on the characteristics of the underlying hardware. The relative size of these subtasks or subunits of computation that are scheduled for parallel execution is referred to as the granularity of the algorithm. Of course, it may be possible to exploit parallelism at more that one level, as in a vector multiprocessor that supports both inner-loop (fine- grain) parallelism in performing larger tasks. The optimal choice of granularity depends on the number and type of processors available and on the overhead associated with communication among tasks, whether by message passing or through shared memory.

We characterize the exchange of information among processors in order to satisfy data dependencies among

subtasks as communication, whether this is accomplished explicitly through message passing or implicitly through shared memory. We have already remarked that many linear algebra problems inherently require global communication, meaning that results produced by a given processor must be made available to all other processors. In a distributed-memory environment, such a communication pattern is typically accomplished by propagating the information in stages through local links in the interconnection network until all processors eventually receive it.

The keys to developing efficient parallel algorithms for a multiprocessor are to maximize concurrency and minimize communication costs. Unfortunately, these two objectives often conflict, and so a compromise must be sought between them. For example, increasing the granularity of an algorithm tends to reduce communication overhead, but it also tends to reduce potential concurrency. The precise trade-off point depends on both the problem and the architecture. The latter can be roughly characterized by the ration of computation speed to communication speed. Both concurrency and communication requirements are determined by the manner in which the computation and the data are partitioned and distributed over the processors. For matrix problems, the main questions are how to organize the computation and partition the matrix how to map the resulting communication requirements.

1.5.2 PARTIONING OF THE MATRIX AND MAPPING

The partitioning of the matrix and mapping of the resulting pieces onto the processors is of critical importance in distributed-memory algorithms because the load balance among processors is determined by this static assignment. Assigning the pieces of the matrix (rows, columns, etc>) to the processors in the same manner that one would deal cards (sometimes called interleaved or scattered storage or wrap mapping) tends to yield better concurrency and load balance in matrix factorization algorithms than assigning contiguous pieces (blocks) to each processor. On the other hand, keeping contiguous data together in each processor tends to reduce necessary communication in these and many other algorithms. This is another example of a trade-off between concurrency and communication that is dependent on the particular problem, algorithm, and machine characteristics.

1.6 RELEVANCE OF PROJECT

Consider the system of linear equations :-

$$AX = B$$

Where A be an n*n real matrix

B be a vector in R**n

X is an unknown vector to be determined.

In solving this system of linear equations matrix inversion is often needed. For example in the above problem be need inverse of matrix A.

There are variety of methods for computing the inverse of matrix. They are classified as direct methods and iterative methods.

Iterative methods :-

--Jacobi's Methods

--Gauss-Seidal Methods etc.

Direct Methods :-

--Gauss Method

--Gauss-Jordan Method etc.

Iterative methods give a sequence of approximate solutions which converge when the number of steps tend to infinity. Thus they do not find an exact solution infinite time , but they converge to solution asymptotically. They are preferred to direct methods when order of matrix n is very

large. Also when matrix A is sparse , they require less storage space than direct methods.

Direct methods produce the exact solution with a finite number of operations generally of the order of n**3 , disregarding the round-off errors.

We will choose Gauss-Jordan algorithm to find the inverse.

We will modify the algorithm to enhance the numerical stability and to reduce the effect of round off or the truncation errors. Then we will develop he parallel algorithm for this sequential algorithm. While implementing parallel algorithm we will apply load balancing to our algorithm to achieve the efficient use of processors i.e. to increase the efficiency or the speedup of the parallel algorithm. If the operating system is not good then efficient application of load balancing may produce the speedup of 60 percent.

Why we have chosen Gauss-Jordan algorithm. There are few points to support it.

- It requires no back subsituion so this is useful particularly in matrix computaions where multiple right hand sides are present.
- It has better scope for loadbalancing and good vectorization property.

- While parallelising vector matrix multiplication is more communication efficient than backward / forward subsitution.

We are using modified Gauss-Jordan algorithm for two reasons:-

1. To enhance the numerical stability which is explaind below

In Gauss-Jordan elimination process there is astep of dividing pivot row by pivot element. In case pivot is zero, then we can not divide the pivot row by it, and process terminates even if the final solution (inverse of matrix) exists. As a remedy we find the element of maximum magnitude of the matrix and call it pivot element. Pivot locations are locations of diagonal elements of matrix . pivot locations for the first location is location of first diagonal element.

Pivot location for second iteration is location of second diagonal element and so on. Now we exchange the rows (with rows) and columns (with columns) to bring pivot element to pivot location.

Since exchanging the columns changes the coefficients of variables. Therefore we keep an account of all column exchanges in some order and while writing final answer we exchange the columns in reverse order to restore the order of variables.

Now the proceed to the steps of Gauss-Jorthan elimination. Since we are lising greatest element of matrix as pivot element, it can never be zero except when every element of matrix is zero. Therefore process will never terminate in between and will produce the inverse of the required matrix if it exists. After completion of all steps of first iteration in elemination process we find new pivot element for second iteration. For this we find the greatest element of the matrix obtained after deleting pivot row & pivot column from the matrix of the previous iteration. After this same steps of elemination process follows.

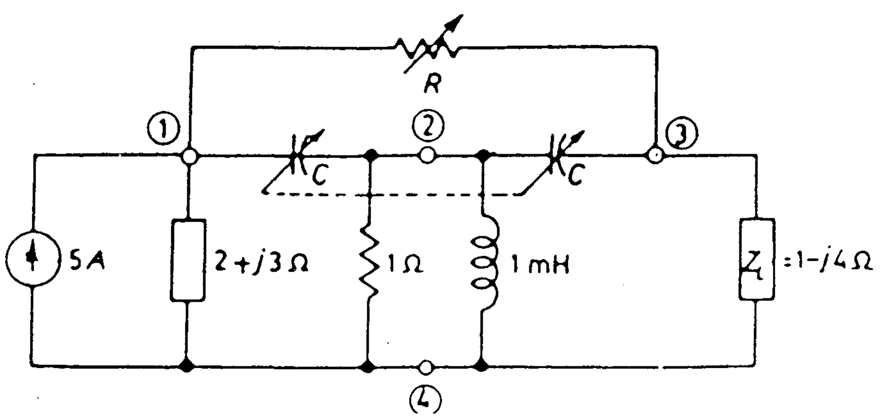
In the first iteration of elemination process we get all element except pivot element of pivot column eleminated. In this way after in iterations (where m is the number of rows) we get only diagonal elements and all other elements are eleminated and we get the inverse directely without substitution.

2. To reduce the effect of truncation/round off errors and make them bounded.

Suppose pivot element is very near to zero, it may contain a large relative error brought about by having to retain a fixed number of digits in each number in the computer. Then dividing by this element will increase the amount of error in each following step and error will increase unbounded. These situations can be remedied by searching at

each iteration of elemination forcess for the element of greatest magnitude as explained in the previous case of numerical stability.

In mathematical modeling of various physical problems we often need matrix computations. In most of the cases elements of matrix vary in magnitude with a large amount. Few elements are very small and few very large. In these cases there is good possibility of selection of very small pivot element. This will introduce large amount of relative round-off errors and a great deal of numerical instability some time it becomes impossible to perform further computations after few since round-off errors multiply very fast in each following step. For example in analysis electrical RLC networks some times we have capacitance of the order of microfarad or picofarad, resistance of the order of Kilohms and inductance of the order of milihenry. In the modelling of these different magnitude and order in the node equations matrix.



The node equations in matrix form can be written as

$$\begin{bmatrix} \left(\frac{1}{2+j3}+\frac{1}{R}+jwc\right) & -jwc & -\frac{1}{R} \\ -jwc & \left(2jwc+1+\frac{1}{jw10^{-3}}\right) & -jwc \\ -\frac{1}{R} & -jwc & \left(jwc+\frac{1}{R}+\frac{1}{1-j4}\right) \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}$$

If the value of $R = 2x10^2\ \Omega$

$W = 100$ rad/sec.

$C = 3x10^{-12}$ F

The order of magnitude of elements will be very much different and we will not be able to find the inverse of matrix with sufficient accuracy and reliability.

Consider another illustrative example which will show the disaster occurred due to selection of small pivot :

Consider the simple system

$$0.000100\ x + y = 1 \qquad\qquad x = 1.00010$$

Solution :

$$x + y = 2 \qquad\qquad y = 0.99990$$

to be solved using three-digit arithmetic.That is, only the three most significant decimal digits of any of any number are retained as the result of an arithmetic operation. We assume the result is rounded. With Gauss Jordan elimination we multiply the first equation by - 10,000 and add to obtain

$$0.000100\,x + y = 1 \qquad x = 0.000$$

Solution :

$$-10{,}000\,y = -10{,}000 \qquad Y = 1.000$$

Thus a computational disaster has occurred.

If we switch the equations (that is, we pivot) to obtain

$$x + y = 2$$

$$0.000100\,x + y = 1$$

TH-6868

then Gauss Jordan elimination produces the system (again with three-digit arithmetic)

$$x + y = 2 \qquad x = 1.00$$

Solution :

$$y = 1 \qquad y = 1.00$$

This solution is as good as one would hope for using three- digit arithmetic.

The lesson of this example is : It is not enough just to avoid zero pivots, you must also aviod relatively small ones.

CHAPTER 2

ARCHITECTURE AND NETWORK TOPOLOGIES

ARCHITECTURE AND NETWORK TOPOLOGIES :-

2.1 ARCHITECTURE

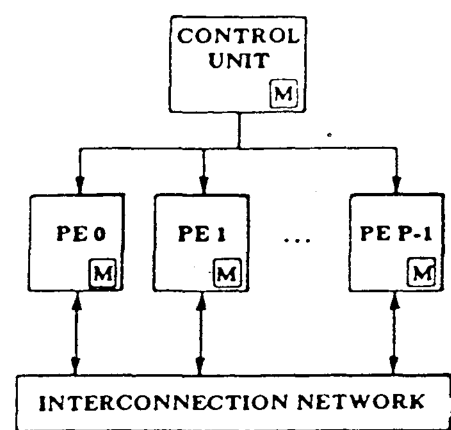
Computer architectures are divide into three groups called SISD, SIMD, and MIMD. SISD (single instruction, single data) architecture explains the operation of classical machines. It uses hardware in a very inefficient way; however, this architecture is simple and easy to program, and most of the computer systems presently available follow this architecture. This is especially true in small systems, and in measurement and control systems. In these systems, memories are relatively small and efficiency is not crucial.

SIMD (single instruction, multiple data) architecture offers more efficient use of hardware but also provides a substantially higher speed of operation. SIMD architecture is presented. an SIMD system typically consists of identitical processing elements (PE), each a processor with its own memory, an interconection network, and a control unit (CU). The control unit broadcasts instructions to all PEs. Each active PE executes each intstruction on the data in its won memory. The instruction is executed simultaneously in all active PEs. The interconnection network enables the data to be transferred among the PEs.

Because the same operation is performed simultaneously on different data items, SIMD systems are particularly well-

suited for performing matrix and vector operations.

MIMD (multiple instruction stream, multiple data stream architecture enables a number of independent but related programs to be executed concurrently. Concurrent processing is single-level or global form of parallelism, denoting current machine thus uses loosely coupled multiple processors to perform many operations at once. Figure shows MIMD concurrent architecture.



Single instruction, multiple data (SIMD) configuration.

2.1.1 FURTHER CLASSIFICATION BASED ON COMMUNICATION

Parallel architecture can be further divided into two categories, based on the way the processors communicate. According to Bell, 1 all of the machines use either bus- based or nonbus-based architectures. Within the bus-based groups are

two subcategories: tightly coupled and loosely coupled systems. The tightly coupled systems, sometimes called multiprocessors and common, or global, memory. The processor and memory are connected by one or more high speed buses. Loosely coupled systems, sometimes called multicomputers, have local memories for each processor, although they sometimes have global memory for shared data.

2.2 NETWORK TOPOLOGIES

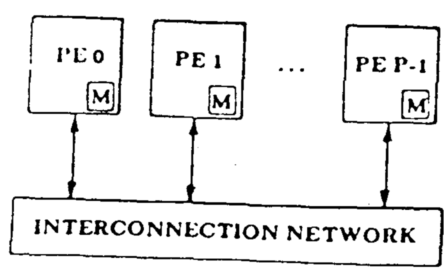
We will represent a communication network of processors as a graph G = (N,A), also referred to, somewhat loosely as a topology. The nodes of the graph correspond to the processors, and the presence of an (undirected) arc (i,j) indicates that there is a direct communication link that serves as an error free, asynchronous packet pipe between processor i and processor j in both directions. We assume that communication can take place sumulsaneously on all of the incident links of a node and in both directions.

We now consider a number of specific topologies.

2.2.1 COMPLETE GRAPH

Here there is a direct link between every pair of processors. Such a network can be implemented by means of a bus which is shared by all processors, or by means of some type of crossbar switch. Clearly this is an ideal network in

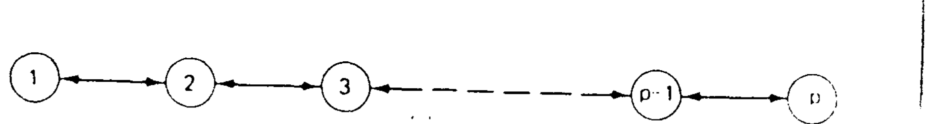
terms of flexibility. Unfortunately, when the number of procssors is very large, a crossbar switch becomes very costly, and a bys involves large queueing delays.



Multiple instruction, multiple data (MIMD) configuration.

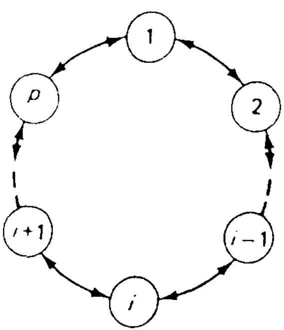
2.2.2 LINEAR PROCESSOR ARRAY

Here there are p processors/nodes numbered 1,2,.......,p and there is a link (i,i+1) for every pair of successive processors . The diameter and connectivity properties of this network are the worst possible.



2.2.3 RING

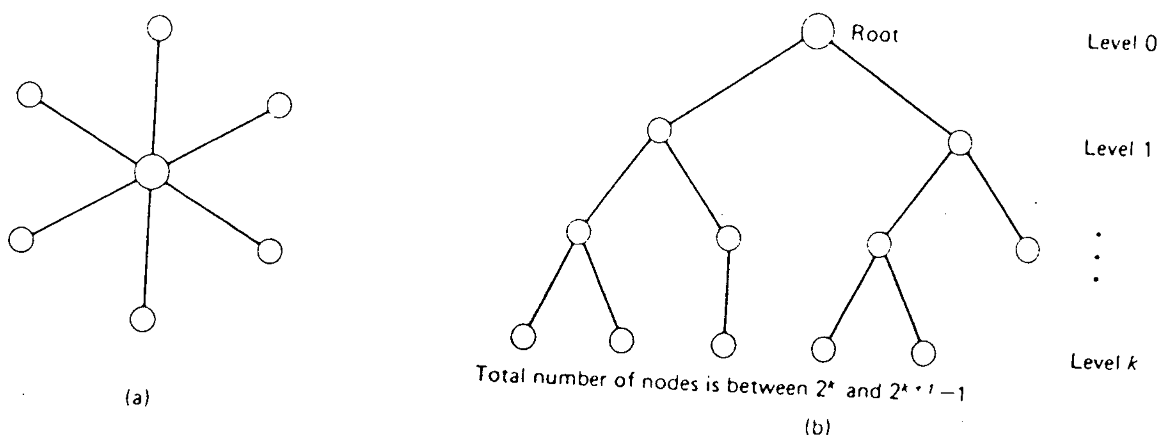
This is a simple and common network that has the property that there is path between any pair of processors even after any one communication link has failed. However, the number of links separating a pair of processors can be as large as [(p-1)/2], where p is the number of processors. It can be seen that all of the basic communication problems discussed earlier (single node and multinode broadcast, single node scatter, and total exchange) can be solved on a ring in a time that lies between the corresponding time on a linear array with the same number of nodes, and one-half that time.



2.2.4 TREE

A tree network with p processors provides communication between every pair of processors with a minimal number of links (p-1). One disadvantage of a tree is its low connectivity; the failure of any one of its links creates two

subsets of processors that cannot communicate with each other. The star network has munimal diameter among tree topologies; however the central node of the star handlkes all the network traffic, and can become a bottleneck.



(a)
(b)

2.2.5 MESH

Many large problems of interest are closely tied to the geometry of physical space. Mesh connected processor arrays are often well suited for such problems. In a d- dimensional mesh the processors are arranged along the points of d-dimensional space that have integer coordinates, and there is a direct communication link between nearest neighbors.

2.2.6 HYPERCUBE

Consider the set of all points in d-dimensional space with each coordinate equal to zero or one. These points may be thought of as the corners of a d-dimensional cube. We let these points correspong to processors, and we consider a communication link for every two points differing in a single

coordinate. The resulting network is called a hypercube or d-cube. shows a 3-cube and a 4-cube.

Formally, a d-cube is the d-dimensional mesh that has two processors in each dimension, that os, ni= 2 for all i. To visualize better a d-cube, we assume that each processor has an identity number which is a binary string of length d (corresponding to the coordinates of a node of the d-cube). We can construct a hypercube of any dimension by connecting lower-dimensional cubes, starting with a 1-cube. In particular, we can start with two (d-1)-dimensional cubes and introduce a link connecting each pair of nodes with the same identity number.This constructs a d-cube with the identity number of each node obtained by adding a leading 0 of leading 1 to its previous identity, depending on whether the node belongs to the first (d-1)-dimensional cube or the second .

The Hamming distance between two processors is the number of bits in which their identity numbers differ. Two processors are directly connected with a communication link if and only if their Hamming distance is unity, that is, if and only if their identity numbers differ in exactly one bit.

Dimensions	Nodes	Channels	Topology
0	1	0	
1	2	1	
2	4	4	
3	8	12	
4	16	32	
5	32	80	

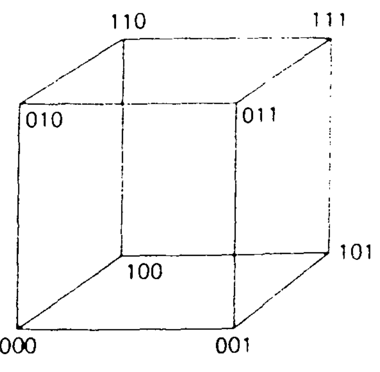
The main architecture in the nonbus-based category is the hypercube. Instead of relying on buses, hypercubes rely on direct memory-access channels between neighbouring processors and their memories. Each processing unit, called a node, can communicate directly with its nearest neighbours in the n-dimensional space in which it was designed and built. The hypercube is a binary n-cube, also referred to as a binary hypercube or boolean hypercube.

Characteristics

A two-dimensional hypercube has four nodes, each at a corner of a single square. Each node is able to communicate directly with two other nodes. A three-dimensional hypercube is the familiar cube. This hypercube has eight noes, each at one corner of cube; each node communicates directly with three other nodes. Higher-dimensioned cubes are built up from this basic structure.

The cube "dimension" equals the power of two corresponding to the number of nodes in the cube. Thus, a 32 node cube is a five-dimensional system (205). Each node is connected to its five nearest neighbors. If the processor needs to communicate with a node that is not one of its nearest neighbors, the data must be routed via intervening processors. If this occurs frequently, it could slow overall processing rates in hypercubes. Each node is a powerful processor operating independently of others. Hence, the

hypercube represents a loosely coupled, coarse-grain architecture.



(a)

0110
1110
1010
0010
1111
0011
0111
1011
0100
1000
1100
0000
1101
0101
0001
1001

(b)

2.3 COURSE GRAIN & FINE GRAIN SYSTEM

In any parallel computer architecture, there is a trade-off between the numbers and the size of the processors. Possible solutions are single-grain systems, coarsegrain systems, and fine-grain systems. A single-grain system is a classical Neumann machine with only one processor. A coarse-grain system couples a moderately large number (say hundreds) of processors. for example, IBMs GF11 machine contains 576 processing elements, interconnected by a three stage switching network; the Connection Machine from Thinking Machines Corporation contains 64,000 processing elements, interconnected by a programmable switching network. Each processor is a simple one-bit processing unit.

2.4 CHOICE OF SYSTEM & ARCHITECTURE

Parallel systems can be built in many different ways; the choice of architecture depends on the application. Many systems exist in each architecture category.

2.5 COMMUNICATION ASPECTS OF PARALLEL AND DISTRIBUTED SYSTEMS

In many parallel and distributed algorithms and systems the time spend for interprocessor communication is a sizable fraction of the total time needed to solve a problem. In this case we say that the ;algorithm experiences substantial

communication penalty or communication delays. We cal think of the communication penalty as the ratio

$$Cp = \frac{T_{TOTAL}}{T_{COMP}}$$

Where T_{TOTAL} is the time required by the algorithm to solve the given problem, and T_{COMP} is the corresponding time that can be attributed just to computation, that is time that would be required if all communication were instantaneous.

To analyze communication issues, it is helpful to view the distributed computing system as a network of pr90cessors connected by communication links. Each processor uses its own local memory for storing some problem data and intermediate algorithmic results, amd exchanges information with other processors in groups of bits called packets using the communication links of the network.

Communication delays can be divided inot four parts:

(a) **Communication processing time:** This is the time required to prepare information for transmission.

(b) **Queueing time:** Once information is assembled into packets for transmission om some communication link, it must wait in a queue prior to the start of its transmission for a number of reasons. for example, the link may be temporarily unavailable because other information packets or system control packets are using.

(c) **Transmission time:** This is the time required for

transmission of all the bits of the packet.

(d) **Propagation time:** This is the time between the end of transmission of the last bit of the packet at the transmitting processor, and the reception of the last bit of the packet at the receiving processor.

Depending on the given system and algorithm, one or more of the above times may be neglibible. For example, in some cases the information is generated with sufficient regularity and the transmision resources are sufficiently plentiful so that there never is a need for queueing packets, whereas in other cases the physical distance between transmitter and receiver is so small that propagation delay is negligible.

For most systems, we can reasonbly assume that the processing and propagation time on a given link is constant for all packets, and the transmission time is proportional to the number of bits (or length) of the packets. We thus arrive at the following formula for the delay of a packet in crossing a link:

$$D = P+RL+Q$$

where P is the processing and propagation time,R is the transmission time required for a single bit, L is the length of the packet in bits, and Q is the queueing time. In the great majority of presently existing systems, even when the packet does not contain much more than overhead, the sum P+RL

is much larger than the time required to execute an elementary numerical operation such as a floating point multiplication. This means that if a prallel algorithm requires transmission of a packet for every few numerical operations it performs, the communication time is likely to dominate its execution time.

Some of the most important factors that influence communication delays are the following:

(a) The algorithms used to control the communication network, mainly error control, routing, and flow control.

(b) The communication network topology, that is, the number, nature, and location of the communication links.

(c) The structure of the problem solved and the design of the algorithm to match this structure, including the degree of synchronization required by the algorithm.

CHAPTER 3

DEVELOPMENT OF PARALLEL ALGORITHM FROM SEQUENTIAL ALGORITHM

DEVELOPMENT OF PARALLEL ALGORITHM FROM SEQUENTIAL ALGORITHM

3.1 HOW TO ACHIEVE PARALLELISM

Parallelism is best used for programs that require a significant number of cycles.

With some body of instructions being repeated a million times or more, we have an opportunity for parallelism if we can spread those million executions in some way across N processors. this is a simple recipe to achieve parallelism:

1. Analyze the program for a loop or recursion structure;
2. Find the instructions that account for the most time, usually the regions repeated the greatest number of iteratioins;
3. Split the instruction execution of these regions across N processors, if this can be done correctly; and
4. Add sycnchronization and data-transmission statements as required to create a correct parallel implementation.

3.1.1 LARGE PROBLEMS

In spite of the continual efforts and achievements in increasing the speeds of logic circuits, memories, and input/output equipment, requirements for processing and computation capabilities have increased at a somewhat comparable rate.

If the magnitude of these large processing and computatioin problems results from a need for processing a number of different sets of data within a given time, one approach to a solution is to use different processing units to work simultaneously on different data sets rather than sequentially forcing different datasets through the same processor in time sequence . Examples of such problems include the global weather problem, nuclear physics problems, large hydrodynamic problems, and others in which an array or mesh of data points are processed.

3.1.2 PROBLEMS WITH INHERENT PARALLELISM

Such inherent parallelism may result from the presence of several data streams which can be processed in parallel, and subsequently under the control of a single instruction streams. Another type of inherent parallelism results from the operations on different sets of data. If such parallel computations are quite frequent in the program, the overall program may be executed more efficiently (from an equipment and economic standpoint) as well as in a shorter elapsed time by the use of some type of parallel organization.

Parallel data streams are typically found in phased array radar, sonar and radar signal processing, and pattern recognition problems.

3.1.3 RELIABILITY AND GRACEFUL DEGRADATION

A parallel organizatioin can permit graceful degradation of the system by dropping off less critical operations in the event of the malfunction of a part of the system, while permitting the more critical operations to continue in those parallel units which are still operable. The failure of a single unit would decrease the total system capability by a very small percentage and would have an almost negligible effect on the necessary functions in the overall system. This is particularly true if the system is designed such that it can automatically detect a malfunction in one unit and reconfigure the problem to perform the critical functions on the remaining units.

3.1.4 PREPARATION AND EVALUATION OF COMPUTER PROGRAMS OF PARALLEL PROCESSING SYSTEMS

The modeling of computations for prallel processing can be divided inot four general areas (1) those general models concerned mostly with formal aspects of prallel processing without regard to actual programming considerations, (2) models that incorporate new programming languages designed to enchance prallel processing, (3) models that incorporate extensions of existing sequential programming languages, and (4) models that attempt to detect and represent parallelism in existing sequential languages.

Each approach has considerable merit. the general models (1) allow us to gain deeper insight into the nature of parallel processes and to prove the validity of such proceses. At this point in time, new languages for parallel processing are embryonic. We need additional information and experience with parallel processing to be able to state just which aspects of prallelism need to be explicityly stated and which should be left to machine (compiler) recognition. Experience with such languages will also teach us the most "natural" means of expression of parallelism.

At present, extensions to existing languages are useful in reformulating existing programs, however such extensions usually require complete recoding of existing programs. Models which attempt to detect parallelism in existing sequential languags can play an important role in this transition period. Large existing programs can be made operational on existing (or simulated) prallel processing systems, and observatioins of system behaviour can be made while reformulation of the program is in process. Nor have we learnied all there is to know about progrms for sequential processors. Some of the modeling for parallel processing has given insight into optimization techniques for single processors.

3.2 PARALLELIZATION

In this chapter firstly we will provide modified Gauss Jordan algorithm to find the inverse of matrix. Then we will find the steps of algorithm which are mutually independent. We can do these steps in parallel saving a great amount of time. We will find these steps which can be done simultaneously if more than one processors are used. We will group these steps into together which will form subsections of algorithm. These subsections are mutually dependent and can't be parallelised. We will develop the parallel algorithm with each subsectionwise. Thus our algorithm is broken into number of subsections.

Although we will use load balancing to adjust the load when numbers of processors is changed, yet we will give parallel algorithm for two cases:

a) number of processors p = m = number of rows of matrix.

b) number of processors p = m x 2m = number of elements in augmented matrix.

In second case number of processors is not limited and we get full parallelisation we will give a subsection of sequential algorithm, then parallel algorithm for case (a) p = m will be developed. After that for case (b) p = m x 2m parallel algorithm will be developed. After that second subsection of sequential algorithm will be parallelised and so on.

We have to find the inverse of a (m x m) matrix A.

Consider the system of linear equations:

A X = I

where

X is unknwon square matrix of order (m x m)

I is unity matrix of order (m x m)

In matrix form:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1M} \\ a_{21} & a_{22} & \dots & a_{2M} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mm} \end{bmatrix} \cdot \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ x_{m1} & x_{m2} & \dots & x_{mm} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ .. & .. & \dots & .. \\ .. & .. & \dots & .. \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

If we are able to find matrix X, it will be inverse of matrix A. We will find matrix X by using elimination process.

Writing the right hand side coefficient matrix.

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ .. & .. & \dots & .. \\ .. & .. & \dots & .. \\ 0 & 0 & \dots & 1 \end{bmatrix} \text{as} \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \dots & \dots & \dots & \dots \\ b_{m1} & b_{m2} & \dots & b_{mm} \end{bmatrix} = B$$

Writing Augmented matrix using matrix A coefficients and right handside coefficients:

Augmented matrix:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} & b_{11} & b_{12} & \cdots & b_{1m} \\ a_{21} & a_{22} & \cdots & a_{2M} & b_{21} & b_{22} & \cdots & b_{2m} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} & b_{m1} & b_{m2} & \cdots & b_{mm} \end{bmatrix}$$

This is a (m x 2m) matrix having rows = m, columns = 2m

We can assume that, this matrix has two parts:

First half (m x m) square matrix has = m rows and first m column

Second half (m x m) square matrix has = m rows and last m columns

Subsection 1

Choose the first element as the greatest element (in magnitude) of the first half matrix and bring it to pivot location. Keep an account of column interchanges.

While choosing pivot element we will search for element of greatest magnitude from the first half of matrix. We will not eliminate coefficients from the second half of matrix because they are coefficients of right hand side variables. We will apply the operations to whole matrix so that right handside coefficients are also modified simultaneously as elimination process proceeds.

Pivot location for 1st iteration = location of 1st diagonal element

Pivot location for 2nd iteration = location of 2nd diagonal element

And so on.

Sequential

```
Z := 1
for z := 1 to m do
begin
    begin
    z := z + 1;
    - var i := 1;
        j := 1; i, j = pivot location;
```

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1M} \\ a_{21} & a_{22} & \dots & a_{2M} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mm} \end{bmatrix}$$

- search 1st row and find row max (1)
- search 2nd row and find row max (2)

.
.
.
.

- search mth row and find row max (m)
- search column made of rowmax (x) and find max (row max (x)).
- pivot := max (row max (x)) = a_{kl}

```
- if (i < > k)
row exchange (i, k);
- if (j < > l) colexchange (j,l);
```

{suppose pivot location = i, j, location of greatest element = k, l. First exchange the row having greatest element with the ith row, then exchange the column containing the greatest element with the jth column}.

```
- c. [100]
```

{where C is an array of record type element. The elements will consist of first and second element. Elements will contain column numbers j & l. We want to keep a record of total column exchanges and do them in reverse order while writing the final answer}

```
- C [y]. first = j;
  C [y]. second = l
end;
```

Parallelisation

Number of processors

Case (a) $p = m \times 2m = 2m^2$

Assume initial each element of matrix is on separate processor:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} & b_{11} & b_{1m} \\ a_{21} & a_{22} & \cdots & a_{2M} & b_{21} & b_{2m} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} & b_{m1} & b_{mm} \end{bmatrix}$$

```
begin
      parbegin
-     Transfer half elements from one subhypercube to another
      subhypercube.
      parend;

      parbegin
-     Compare each pair and retain the greater element in that
      node and reject the another one.
      parend ;

      parbegin
-     Transfer quarter elements (half of remaining) from one
      subhypercube to anther subhypercube.
      parend;
      .
      .
      .
      .
-     compare the last two elements and find greatest elements
      a_kl
-     i, j = pivot location
      i := 1; j := 1
-     if (i < > k) row exchange (i, k);
-     if (j < > l) col exchange (j, l);
-     c[y]. first = j
-     c[y]. second = l

end,
```

Case (b)

Number of processors $p = m$

Assume initiality

1st row of matrix on processor P_0

2nd row of matrix on processor P_1

:

:

mth row of matrix on processor P_{m-1}

begin

parbegin

- search 1st row on P_0 for max (row 1)

- search 2nd row on P_1 for max (row 2)

:

:

- search row on P_{M-1} for max (row m)

parend;

parbegin

- Transfer half of max (row x) from one subhypercube to another subhypercube.

parend;

parbegin

- compare two elements of max (row x) on each node. Retain the greater.

parend;

parbegin

- Transfer half of max (row x) elements from one subhypercube to another subhypercube

parend;

```
:
:
    -    compare the last two max (row x) and retain max
         (max (row x)) = a_kl
    -    i, j = pivot location
         i := 1; j := 1;
    -    if (i < > k) row exchange (i, k);
    -    if (j <> l) colexchange (j, l)
    -    C [y]. first = j
    -    C[y]. second = l
end,
```

Subsection 2

Divide each element of pivot row by pivot element.

Sequential

```
for j = 1 to 2 m do
begin
     a_ij div pivot element
     j := j + 1
end;
```

Parallel

Case (a)

```
begin
     parbegin
     -    Send (transfer) pivot element to each 2 m
```

processors using single node broadcast

parend

parbegin

- Tranfer each element of 2 m pivot row elements on each individual processors (already exist)

parend

parbegin

- divide a_{i1} on P_0 by a_{ii} (pivot element)
- divide a_{i2} on P_1 by a_{ii}
- divide a_{i3} on P_2 by a_{ii}

:

:

- divide b_{i1} on P_m by a_{ii}
- divide b_{im} on P_{2m-1} by a_{ii}

parend

end;

{after execution of this subsection pivot row is modified}

Case (b)

p = m

begin

parbegin

- Transfer first m elements of pivot row on m separate processors: m times single node broadcast.

parend;

parbegin

- Transfer pivot element to each m processor.

Wing single node broadcast.

```
        parend;
        parbegin
        -    Divide each element on processors by pivot element.
        parend;
        parbegin
        -    Transfer last m elements of pivot rows on m
             separate processor: m times single node broadcast.
        parend;
        parbegin
        -    divide each element on processor by pivot element.
        parend;
end;
```

{After this subsection each processor has two modified pivot row elements

eg. P_0 has a_{i1}, b_{i1}

P_1 has a_{i2}, b_{i2} and so on}.

Subsection 3

Keep the m copies of modified pivot row and multiply each of them by appropriate factors separately.

Sequential

```
begin
        begin
        -    Load modified pivot row on processor
        -    multiply each element of this row by - a21
        -    store the results.
```

```
end;
     begin
     -    Load modified pivot row on processor
     -    multiply each element of this row by - a31
     -    store the results
end;
     :    and so on for all (m-1) rows.
     :
end;
```

Parallel

Case (a)

```
begin
     parbegin
     -    Transfer all 2m elements of modified (after
          division) pivot row through columns using single
          node broad cast for all 2m nodes.
     parend;
          {now modified pivot row has m copies}
     parbegin
     -    Transfer - a21 to each element of 2nd row
     -    Transfer - a31 to each element of 3rd row.
     :
     :
     -    Trnasfer - am1 to each element of mth row
     parend;
     parbegin
```

- Multiply 2nd row of processor by - a_{21}
- Multiply 3rd row of processor by - a_{31}

:

- Multiply mth row of processor by - a_{m1}

parend;

end;

{After this subsection modified rows elements after multiplication are on the same processors on which original elements are present}

Case 3(b) p = m

{Keep the copies of modified pivot row after division on m processors}

begin

parbegin

- Transfer the each of two elements in each processor to each processor using multinode broadcast two times.

parend;

{Now - a_{21} is present in P_1

- a_{31} is present in P_2 and so on}

parbegin

- multiply each element on P_1 by - a_{21}
- multiply each element on P_2 by - a_{31}
- multiply each element on P_3 by - a_{41}

:

:

```
        -    multiply each element on P_{m-1} by - a_{m1}
        parend
end;
```

Subsection 4

{After multiplication elements of rows on processors have been modified and original row elements corresponding to them are on same processors}

Sequential

```
begin
        -    Add corresponding elements of modified & original
row R2
        -    Add corresponding elements of modified & original
             row R3
        :
        :
        -    Add corresponding elements of modified & original
             row Rm.
end
        for i := max down to 1 do
        col exchange (c[y]. first, c[y]. second)
        -    delete pivot column
end;
        {of for loop Z := 1 to m do}
```

Parallel Case (a) $p = 2m^2$

```
parbegin
     parbegin
     -    Add corresponding elements of modified and original
          row R2
     parend
     parbegin
     -    Add corresponding elements of modified and original
          row R3
     parend
     .
     .
     .
     parbegin
     -    Add corresponding elements of modified and original
          row Rm
     parend
     parend;
     for i := max down to 1 do
     col exchange (c[y].first, c[y].second)
     -    delete pivot column
     parend
end;
```

Case (b) $p = m$

```
begin
     parbegin
     -    Add corresponding elements of modified row on P1 and
          2nd original row R2
```

- Add corresponding elements of modified row on P_2 and 3rd original row R_3
- Add corresponding elements of modified row on P_3 and 4th original row R_4

.
.
.

- Add corresponding elements of modified row on P_{m-1} and mth original row R_m

parend

end;

for i := max down to 1 do

col exchange (c[y]. first c[y]. second)

- delete first column

end;

{After execution of above these 4 subsections of algorithm our augmented matrix looks like -

$$\begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1M} & b'_{11} & \dots & b'_{1m} \\ 0 & a'_{22} & \dots & a'_{2M} & b'_{21} & \dots & b'_{2m} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & a'_{m2} & \dots & a'_{mm} & b'_{m1} & \dots & b'_{mm} \end{bmatrix}$$

First column except pivot element of this augmented matrix has been eliminated. In the second iteration of these 4 sections 2nd column wll be eliminated. For the second iteration we will choose pivot element from the submatrix obtained after deleting 1st column and 1st row that is pivot column and pivot row from the first half sq. matrix.

After completion of final iteration we will get only diagonal elements and all others will be eliminated from the first half of sq. matrix. Thus from the direct substitution we will get the required inverse of matrix of A from the second half of the augmented matrix.

CHAPTER 4

COMPLEXITY ANALYSIS

COMPLEXITY ANALYSIS

4.1 COMPLEXITY OF ALGORITHMS

In order to measure the cost of executing a program,we customarily define a complexity (or cost) function F, where F(n) is a measure of the time required to execute the algorithm on a problem of size n, or a measure of the memory space required for such execution. Accordingly, we speak of the time complexity and the space complexity functions of the algorithm.

In practice, of importance is performance of the algorithms for large values of n, that is performance of the angorithms for large values of n, that is asymtotic behaviour of the complexity function. Asymptotic complexity is the growth in the limit of the complexity function with the size parametern. So the asymptotic (time or space) function ultimately determines the size of the problem that can be solved by the algorithm. In terms of the terminology, we say that the (time) complexity of the algorithm is O(log n), read 'order log n' if the processing by the algorithm of the problem instance of size n takes the time proportional to log n. In relation to, say the time compexity of an algorithm the following terminology will be used in an equivalent sense:

(a) the time compexity of the algorithm is of order log n; this can also be written as 'O(log n)';

(b) the algorithm is executed is executed in O(log n) time;

(c) the amount of work require by the algborithm is proportionalto log n, or is of O(log n).

Where appropriate the term 'unit of time' will be used in the sense equivalent to the term 'one basic operation'

4.2 BOUNDS ON COMPLEXITY

Complexity analysis among other things is concerned with obtaining upper and lower bonds on the performance of algorithms or classes of algorithms that solve various problems. The existence of complexity bounds for the known algorithms can serve as a basis for classifying the problems. for other problems lower bounds on complexity have been derived but none of the available algorithms is known to attain the bounds. A notable example of this type is matrix multiplication, where a minimum bound of O (n^2) is not sharp enough.

Another group of problems is such that their lower bounds on computational complexity are known, but these algorithms are numerically unstable. Fast methods approach of interpreting the operation of matrix multiplications belong to this category.

Final, there are problems for which lower bounds on complexity are known and the algorithms which attain these bounds can be built and the algorithms are numerically stable.

4.3 CONDITIONING OF THE PROBLEM

Consider the set

$$Ax = b \qquad (1)$$

Let $x = A_{-1}b$ be the exact solution of the set and $x_{(c)}$ be its computed solution. We shall distinguish between the difference

$$A^{-1}b - x^{(c)} = x - x^{(c)} \qquad (2)$$

Which is called the error and the difference

$$b - Ax^{(c)} \qquad (3)$$

Which is called the residual.

Studies on the conditioning of problem 1) show that if both the error and the residaual vectors are small then the problem is well-conditioned. If the residual is small while the error is very large then the problem is called ill-conditioned. The matrix of an ill-conditioned is unsually 'nearly singular' or even exactly singular.

If an instance of problem 1) is ill-conditioned then no computational algorithm will solve it accurately. For a well-conditioned problem one expects that a reasonable approximation to the exact solution can be computed if a stable computational algorithm is used. Numerical stability of a particular algorithm is studied using the concept of the error as defined by 2), of of the residual as defined by 3). If the solution, $x^{(c)}$, is computed using a particular algorithm is numerically stable and vice versa.

In the forward error analysis (of pre-Wilkinsin era) one attempts to bound the difference between the exact and computed solutions at every step of the computation. As the computation progresses this becomes more and more difficult, the bounds on the difference between the solutions become 'loose'' and this leads to far too pessimistic conclusions on the method's relaiability. The Wilkinson's backward error analysis, on the other hand, studies the residual and not the error. The approach is simpler and gives reasonably sharp bound estimates on the residual and not the error. The approach is simpler and gives reasonably sharp bound estimates on the residual which in turn leads to more realistic conclusions on the reliability of the method.

4.4 SPEEDUP AND EFFICIENCY

We describe a few concepts that are sometimes useful in comparing serial and parallel algorithms. Suppose that we have a parallel algorithm that uses p processors (p may depend on n), and that terminates in time $T_p(n)$. Let $T^*(n)$ be the optimal serial time to solve the same problem, that is, the time required by the best possible serial (uniprocessor) algorithm for this problem. The ratio

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

is called the speedup of the algorithm, and describes the speed advantage of the parallel algorithm, compared to the best possible serial algorithm. The ratio

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{pT_p(n)}$$

is called the efficiency of the algorithm and essentially measures the fraction of time that a typical processor is usefully employed.Ideally , $S_p(n) = p$ and $E_p(n) = 1$, in which case, the availabity of p processors allows us to speed up the computation by a factor of p. For this occur,the parallel algorithm should be such that no processor ever remains idle or does any unnecessary work. This ideal situation is practically unattainable. A more realistic objective is to aim at an efficiency that stays bounded aqay from zero,as n and p increase.

There is difficulty with the above difinitions because the optimal serial time $T^*(n)$ is unknown,even for seemingly simple computational problems like matrix multiplication. For this reason, $T^*(n)$ is sometimes defined differently.

We may let $T^*(n)$ be the time required by a single processor to execute the particular parallel algorithm being analyzed. (That is,we let a single processor simulatew the operation of the p parallel processors,). With this choice of $T^*(n)$, efficiency relates to how well a particular algorithm has been parallelized.

4.5.1 NUMERICAL ACCURACY

In the analysis of algorithms which solve numerical

problems the accuracy of the computed reesults is another important criterion to distinguish between 'good' and 'bad' algorithms. The need to examine the accuracy of mathematical computations arises from the fact that a computer is a finite machine; it is capable of repreaenting numbers only to a finite number of digit positions. As a result, most numbers, and even integers, if they are too long for the computer to reprtesent exactly, are rounded, and so only a finite approximation some numerical algorithms implemented on a computer may produce approximations to the true results that are wildly inaccurate.

4.5.2 EFFICIENCY

Peak performance is very special state that is rarely achievable. There are several factors that introduce inefficiency. Among the factors are:

* The delays introduced by interprocessor communications;
* The overhead in synchronizing the work of one processor with another;
* Lost efficiency when one or more processors run out of tasks;
* Lost efficiency due to wasted effort by one or more processors;
* The processing costs for controlling the system and schedulingoperations.

A high-performance vector processor is suffer from lost performance because it is unable to keep all of the processing units busy. This latter problem arises particularly when a computation is not easily implemented as a sequence of vector operations performed on highly structured, densely stored data.

The architect who designs and builds a multiprocessor must pay close attention to the sources of inefficiency exposed here. They can lead to serious degradation in performance. For example, if the combined inefficiencies produce an effective processing rate of only 10 percent of the peak rate, then ten processors are required in a multiprocessor system just to do the work of a single processor.

Fortunately, for a small number of processors, careful design can hold the inefficiency to a low figure, but inefficiencies tend to climb as the number of processors increase. There is a point where adding additional processors can lengthen, not shorten, computation time.

The fact that inefficiency tends to grow with the number of processors is the underlying reason why many commerical offerings of multiprocessors have a small number of processors, such as 4,8, or 16. The fastest machines are built from the fastest devices available and have relatively few processors.

Consider, for example, the Cray XMP, a four-processor version of the Cray I. Another example is the IBM 309X family for which systems with up to six processors are available.

4.5.3 RUNTIME AND COMMUNICATION OVERHEAD

The point of this section is to analyze the performance benefit of multiple processors in the face of overhead incurred to create parallelism. This section shows that performance benefits strongly depend on the ration R/C, where R is the length of a run-time quantum and C is the length of communications overhead produced by that quantum. The ratio expresses how much overhead is incurred per unit of computation. When the ratio is very low, it becomes unprofitable to use parallellism. When the ratio is very high, parallelism is potentially profitable. Note that a large ratio can be obtained by partitioning a computing job into relatively few large pieces, and that the amount of parallelism for such a ratio might be much smaller than the maximum available.

The ratio R/C is a measure of task granularity:

* In coarse-grain parallelism, R/C is relatively high, so each unit of computation produces a relatively small amount of communication : and
* In fine-grain parallelism, R/C is very low, so there is a relatively large amount of communication and other overhead per unit of computation.

Coarse-grain parallelism arises when individual tasks are large and over head can be amortized over many computational cycles. fine-grain parallelism usually provides opportunities to perform execution on many more processors than can fruitfully support coarse-grained parallelism. The idea of fine-grain parallelism is to partition a program into increasingly smaller taks that can run in paralle. At the ultimate limit, each individual task may be as small as a single operation. More commonly, however, a fine-grained task contains a small number of instructions.

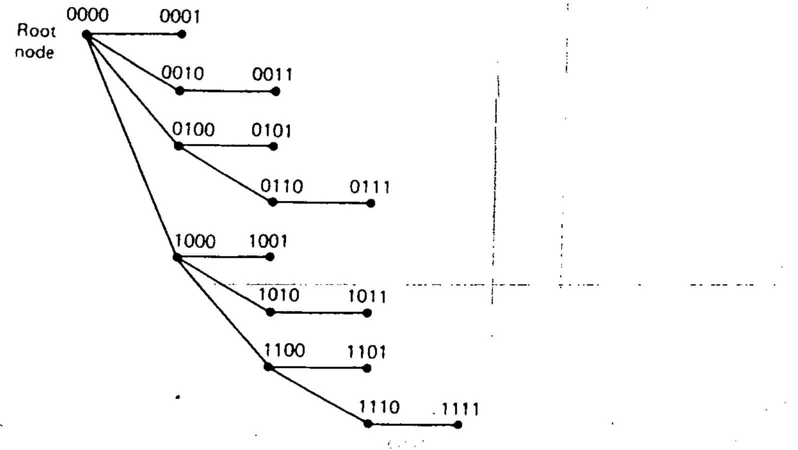
Small R/C ratios lead to poor performance because of high overhead. Large ratios usually reflect poor exploitation of parallelism. For maximum performance, it is necessary to balance parallelism against overhead.

What are good parallel algorithms for solving varioius important poblems? The key approach is the ability to partition the problem intob modules that require relatively little intermodule communication. If the partitioning can be done successfully, then communication requirements are rather small, and the dependency on the interconnection topology is greatly diminished. On the other hand, if communication requirements cannot be made small, then the interconnection topology becomes important, and the major parameter of interest is the R/C ratio.

4,6 CALCULATION OF COMPLEXITIES :

In this chapter we will compare complexities of sequential and parallel algorithms. Although communication cost is negligible in comparism to computation cost, yet we will take it into account. When number of processor increases the communication cost and synchronisation cost come into the picture.

There are two strategies mostly used for communication between nodes. Spaning tree structure is used for these strategies. Spanning tree of a d-cube that is rooted at node (00..0), and provides a path of d links or less from the root node to every other node. The figure shows one possible construction for d = 4. The tree is constructed sequentially starting from the root by using the rule that the identities of the children of each node are obtained by reversing one of the zero bits of the identity of the parent that follows the right most unity bit. The leaf nodes are the ones that have one as the final bit in their identity.



SPANNING TREE

Two strategies are:

1. SINGLE NODE BROADCAST/ACCUMULATION

To send the same packet from a given processor to every other processor is called signle node broadcast. While in single node accumulation, we want to send to a given node a packet from every other node. Using spanning tree a single node broadcast from the root to all nodes, and a single node accumulation take O(d) = O (log p) communication time.

2. MULTINODE BORADCAST/ACCUMULATION

If we want to do a single node broadcast simultaneously from all nodes, we call this a multinode broadcast and simultaneous single node accumulation from all nodes is called multinode accumulation. Both takes a time of

$$O\left(\frac{p}{\log p}\right) = O\left(\frac{m}{\log m}\right) \text{ if } p = m$$

To calculate the complexities assume:

α time taken in 1 additon/subtraction

β time taken in 1 multiplication

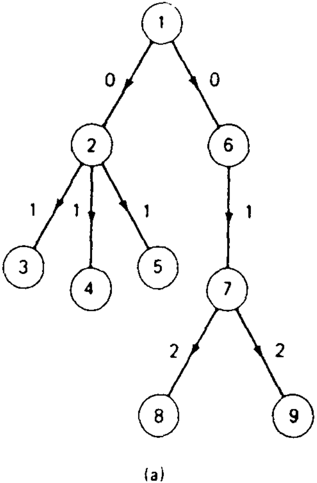
γ time taken in 1 division

δ time taken in 1 comparison

t time taken in transfering unit load from one node to nearest node in hypercube.

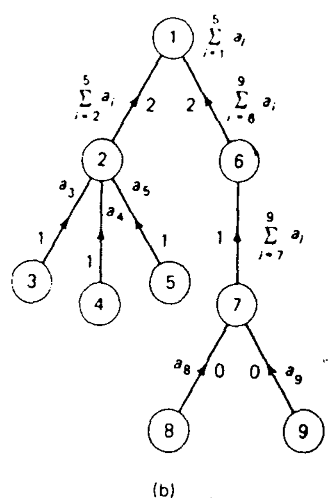
And in sequential processing processor takes negligible time in loading and storing data to memory.

SINGLE NODE BROADCAST



(a)

SINGLE NODE ACCUMULATION



(b)

Also assume

$$0(\alpha) = 0(\beta) = 0(\gamma) = 0(5\delta) = 0(10t)$$

First we will calculate the complexity of sequentila algorithm step by step. We will calculate separately for subsection 1 and for rest of the algorithm because they yield different recursive functions.

SEQUENTIAL COMPLEXITY

Subsection 1

(From the algorithm given in previous chapter)

Time taken in finding row max (i) = $0(m)$

Time taken in finding max (row max (i)) = $0(m) \times 0(m) = 0(m^2)$

After completion of 1st iteration and for selection of pivot element for 2nd iteration we delete one column and one row from the (m x m) square matrix. Thus if initial size of problem = $f_1(m, m)$

then after one iteration size of problem = $f_1(m-1, m-1)$.

Thus,

$$f_1(m, m) = f_1(m-1, m-1) + m^2$$

This is a recursive function on solving this function:

$$f_1(m, m) = f_1(m-1, m-1) + m^2$$

$$f_1(m-1, m-1) = f_1(m-2, m-2) + (m-1)^2$$

$$f_1(m-2, m-2) = f_1(m-3, m-3) + (m-2)^2$$

.
.
.

$f_1(2, 2) = f(1, 1) + 2^2$

$f_1\ (1, 1) = 0$

Adding up $f_1\ (m, m) = m^2 + (m-1)^2 + m-2)^2 + \ldots 2^2$

$= m^2 - 1$

$$= \frac{m\ (m+1)\ (2m+1)}{6} - 1$$

$$f_1(m,m) = 0(m^3) \qquad (1)$$

Subsections 2, 3, 4

Now we will calculate complexity due to subsections 2, 3 and 4.

In first iteration time taken in subsection 2

$= (2m-1)\ \gamma$

Time taken in first iteration in subsecton 3

$= (2m \times \beta)\ (m-1)$

where '2m' due to 2m elements in one row (m-1) due to (m-1) rows to be multiplied.

Time taken in first iteration in subsection 4

$= (2m-1)\ \alpha\ .\ (m-1)$

{first element will be zero after addition and need not to be added}

Total time $= (2m-1)\ \gamma + 2m\ (m-1)\ \beta + (2m-1)\ (m-1)\alpha$

After completion of first iteration first column is zero except pivotal element which is 1 in each iteration. Therefore we can reduce one column from our problem. And our problem

size reduces to f_2 (m′, m−1) from f_2 (m′, m).

{here although m′ = m, but to distinguish between rows and columns we keep them like that}

that is:

$$f_2(m',m) = f_2(m',m-1) + (2m-1)\gamma + (m'-1)\ [2m\beta + (2m-1)\alpha]$$

on solving this recursive function we get:

$$f_2(m',m) = f_2(m',m-1) + (2m-1)\gamma + (m'-1)\ [2m\beta + (2m-1)\alpha]$$

$$f_2(m',m-1) = f_2(m',m-2) + 2(m-3)\gamma + (m'-1)\ [(2m-1)\beta + [2(m-1)-1]\alpha]$$

$$f_2(m',m-2) = f_2(m',m-3) + 2(m-4)\gamma + (m'-1)\ [(2m-2)\beta + [2(m-2)-1]\alpha]$$

:

:

$$f_2(m',2) = f_2(m',1) + 2(2-1)\gamma + (m'-1)\ [(2.2\beta + (2.2-1)\alpha]$$

$$f_2(m',1) = 0 + 2(1-1)\gamma + (m'-1)\ [(2\beta + (2-1)\alpha]$$

Adding up we get

$$f_2m',m = \gamma\left[(2m-1) + [2(m-1)-1] + [2(m-2)] + ..(2-1)\right]$$

$$+ (m'-1)\left[2\beta(m+m-1+...1) + \alpha\left[(2m-1) + [2(m-1)-1] + .. (2-1)\right]\right]$$

$$= \gamma\left[2\left[\frac{m(m+1)}{2} - m\right] + (m'-1)\left[2\beta.\frac{m(m+1)}{2}\right] + 2\alpha\left[\frac{m(m+1)}{2} - m\right]\right]$$

$$= \gamma\left[m^2 + m - 1m\right] + (m'-1)\left[\beta(m^2+m) + \alpha(m^2-m)\right]$$

since m = m′

$$f_2(m,m) = (m^2-m)\ \gamma + (m^3-m)\ \beta + (m^3 - 2m^2 + m)\ \alpha$$

$$f_2(m,m) = 0\ (m^3) \tag{2}$$

From equations 1 and 2 we get over all complexity of sequential algorithm:

$$f(m,m) = f_1\ (m,m) + f_2\ (m,m)$$
$$= 0\ (m^3) + 0\ (m^3)$$
$$f(m,m) = 0(m^3) \tag{A}$$

Parallel Algorithms

Now we will calculate the complexity of parallel algorithm when number of processors p = m, when matrix is of the order of (m x m).

<u>Case (a) p = m</u>

Subsection 1

Time required for searching all rows for max row (i) = 2 m.δ

Time in all transfers = t.log m

Time in all further comparisons = δ.log m

Thus total time taken in first iterations

$$T = 2\ m.\delta + \log m\ \delta + \log m.t.$$

After completion of one iteration problem of size f_1 (m,m) reduces to f_1 (m-1, m-1).

i.e. $f_1(m,m) = f_1(m-1,\ m-1) + \delta\ (2m + \log m) + t.\log m$

on solving this recursive function we get:

$$f_1(m,m) = f_1(m-1, m-1) + 2\delta . m + (t+\delta) \log m$$

$$f_1(m-1,m-1) = f_1(m-2, m-2) + 2\delta (m-1) + (t+\delta) \log (m-1)$$

$$f_1(m-2,m-2) = f_1(m-3, m-3) + 2\delta (m-2) + (t+\delta) \log (m-2)$$

.
.
.

$$f_1(2,2) = f_1(1,1) + 2\delta . 2 + (t+\delta) \log 2$$

$$f_1(1,1) = 0$$

Adding up we get

$$f_1(m,m) = 2\delta \, [m + (m-1) + (m-2) + .. 2] + (t+\delta) \, [\log m + \log (m-1) + \log (m-2) + \ldots \log 2]$$

$$= 2\delta \left[\frac{m(m+1)}{2} - 1\right] + (t+\delta) \, [\log m + \log (m-1) + \ldots \log 2]$$

$$f_1(m,m) = 0 \, (m^2) \qquad (3)$$

Subsections 2, 3 & 4

Subsection 2

Time taken in 1st parbegin - parend statement = $m.\log (m)$

Time in 2nd parbegin - parend statement = $\log (m)$

Time in 3rd parbegin - parend = γ

Time in 4th and 5th statement = $m \log m + \gamma$

Total time = $2m \log m + \log m + 2\gamma$

Subsection 3

Time taken in 1st parbegin - parend statment = $\frac{2.m}{\log m}$

Time taken in 2nd parbegin - parend statement = $2 \, m.\beta$

Total time = $2m \left(\beta + \frac{1}{\log m}\right)$

Subsection 4

Time taken in only parbegin - parend statement = $2\ m.\alpha$

Total time in steps 2, 3 & 4

$$= 2m \log m + \log m + 2\gamma + \frac{2m}{\log m} + 2m\ (\beta+\alpha)$$

$$= (2m+1) \log m + 2m \left(\beta + \gamma + \frac{1}{\log m}\right) + 2\gamma$$

if m is large, $(\beta + \gamma) >> 1/\log m$ and $(2m + 1) = 2m$

Hence {log m will remain constt. since it depends on number of nodes.

$f_2(m,m) = f_2\ (m,m-1) + 2m\ (\log m + \beta + \gamma)$

$f_2(m,m-1) = f_2\ (m,m-2) + 2(m-1)\ (\log m + \beta + \gamma)$

.

.

.

$f_2\ (m,2) = f_2\ (m,1) + 2.2\ (\log m + \beta + \gamma)$

$f_2\ (m.1) = 2.1\ (\log m + \beta + \gamma)$

Adding up we get

$$f_2(m,m) = 2\ (\log m + \beta + \gamma)\ [m + m-1 + \ldots 1]$$

$$= (\log m + \beta + \gamma)\ [m^2 + m] = 0\ (m^2 \log m) \qquad (4)$$

Since $f\ (m,m) = f_1\ (m,m) + f_2\ (m,m)$

$= 0(m^2) + 0(m^2 \log m)$

$$f(m,m) = 0(m^2 \log m) \qquad (B)$$

Case (A)

When number of processor p = 2m x m

Subsection 1

Time in transfer parbegin statements = $t.\log 2m^2$

Time in compare parbegin statements = $t \cdot \log 2m^2$

Total time = $(t + \delta) \log 2m^2$

$f_1(m,m) = f_1(m-1, m-1) + (t + \delta) \log 2m^2$

This is a recursive function. On solving this function we get.

$f_1(m,m) = 2m \log 2m (t + \delta)$

$f_1(m,m) = 0 (2m \log 2m)$ (5)

Subsection 2

Time in 1st parbegin statement = $2 \log 2(m)$

There is no need of 2nd statement

Time in 3rd parbegin statement = γ

Total time = $2 \log 2(m) + \gamma$

Subsection 3

Time in 1st parbegin statement = $2 \log 2m$

Time in 2nd parbegin statement = $2 \log 2m$

Time in 3rd parbegin statement = β

Total time = $\beta + 2.2 \log 2m$

Subsection 4

Time = α

Total time in subsections 2, 3 and 4

$= \alpha + \beta + 2.3 \log 2m - \gamma$

$$f_2\ (m,m) = f_2\ (m,m-1) + (\alpha+\beta+\gamma) + 6 \log 2m$$

In this recursive function log 2m will remain constant since it depends on no. of nodes.

Solving this recursive function we get:

$$f_2\ (m,m) = f_2\ (m,m-1) + (\alpha+\beta+\gamma) + 6 \log 2m$$

$$f_2\ (m,m-1) = f_2\ (m, m-2) + (\alpha+\beta+\gamma) + 6 \log 2m$$

.
.
.

$$f_2\ (m,1) = f_2(m,0) + (\alpha+\beta+\gamma) + 6 \log 2m$$

On Adding up

$$f_2\ (m,m) = m\ (\alpha+\beta+\gamma) + 6\ m \log 2m$$

$$f_2\ (m,m) = 0\ (m \log m) \qquad (6)$$

since $f(m,m) = f_1\ (m,m) + f_2\ (m,m)$

From eqns. (5) and (6) we get:

$$f(m,m) = 0(m \log m) + 0\ (m \log m)$$

$$f(m,m) = 0\ (m \log m) \qquad (C)$$

In equations (B) and (C) the factor of log (m) comes due to communication cost. From this it is clear that if number of processor p = m or m^2 is low then communication cost can be ignored. Summarizing the results in a table:

	Sequential P = 1	Parallel P > 1	
	No. of processor p = 1	p = m	p = $2m^2$
with communication cost	0 (m^3)	0 (m^2 log m)	0 (m log m)
without communication cost	0 (m^3)	0 (m^2)	0 (m)

CHAPTER 5

IMPLEMENTATION AND LOAD BALANCING

IMPLEMENTATION AND LOAD BALANCING

5.1 IMPLEMENTATION

In this chapter we will describe how parallel algorithms are implemented on the subcube grid.

We will give two illustrative models which will explain the principles and rules. The operations described in the model implementation will not necessarily match the actual operations while implementing on the parallel machine. We will take an examples.

No. of processors p = m = 16

Suppose we want to find the inverse of (16 x 16) square matrix A: Augmented matrix will be:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,16} & b_{11} & \cdots & b_{1,16} \\ a_{21} & a_{22} & \cdots & a_{2,16} & b_{21} & \cdots & b_{2,16} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\ a_{16,1} & a_{16,2} & \cdots & a_{16,16} & b_{16,1} & \cdots & b_{16,16} \end{bmatrix}$$

order of the augmented matrix = (16 x 32).

Assume number of processor p = m = 16.
We will take hypercube of dimension = 4 = d.
configured as $r_1 \times r_2$ subcube grid
= 4 x 4 subcube grid.

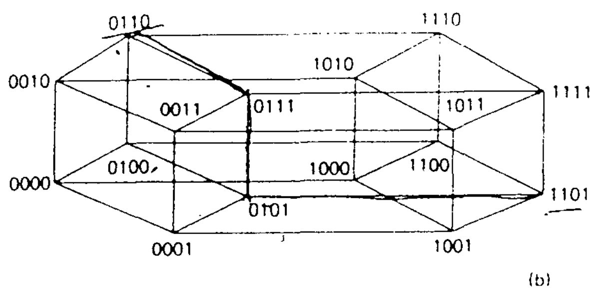
dimension of each subcube column = d_1 = 2

dimension of each subcube row = d_2 = 2

$d_1 = \log \gamma_1 = 2$

$d_2 = \log \gamma_2 = 2$

Number of processor $p = 2^d = 2^4 = 16$



For implementaton of the algorithm given in previous chapter we will use the same subsections of the algorithm.

Subsection 1

Matrix elements are loaded through host processor.

Assume initialy

1st row of matrix is on processor P_0

2nd row of matrix is on processor P_1

.

.

.

16th row of matrix is on processor P_{15}

- Apply linear search to each processor to find row max (i). After time $2m\delta$ each processor has row max (i).
- Transfer data from (0, 2, 4, 6) subhypercube to (1, 3, 5, 7) and from (8, 10, 12, 14) to (9, 11, 13, 15) subhypercube time = t (1 unit).
- In next δ time compare the two elements in nodes and retain the greater.
- Transfer data from subhypercube (1, 3, 5, 7) to subhypercube (9, 13, 11, 15).
- In next δ time we have greater elements in 9, 13, 11, 15).
- Transfer data from (11, 15) to subhypercube (9, 13).
- In next δ time we have greater element in (9, 13).
- Transfer data from 9 to node 13.
- In next δ time we have greatest element in node 13.

Total time = $2\ m\delta + (\delta + t)\ \log m = 0(m)$

Subsection 2

Now suppose pivot row comes out to be in processor P_0

- Load first m elements of pivot row on separate m processors through single node broadcast m times.

  time = m. log (m)

  Now if 1st row is pivot row

  then a_{11} on P_0

  a_{12} on P_1

  a_{13} on P_2

.
.
.

a_{16} on P_{15}

- Send pivot element to each processor using single node broadcast.

  time = log m

- Divide each element staying on separate processor by pivot element

  time = γ

- Load remaining m elements of pivot row on m processors.

  time = m (log m)

- Divide each element by pivot element

  time = γ

  Total time = 2 (m log m + γ) + log m

  = 0 (m log m)

After completion of this subsection pivot row elements are modified and lies on processors such that

a'_{11}, b'_{11} on P_0
a'_{12}, b'_{12} on P_1
and so on.

Subsection 3

- Keep the copies of modified pivot row on m processors using 2 multinode borad cast.

  $$\text{time} = \frac{2\ m}{\log m}$$

Due to this step

+ a_{21} comes in P_1

a_{31} comes in P_2

and so on

- Multiply each element on P_1 by $- a_{21}$
- Multiply each element on P_2 by $- a_{31}$

.
.
.

- Multiply each element on P_{m-1} by $- a_{m1}$

Time = $2m \times \beta$

Total time = $2\ m\beta + \frac{2\ m}{\log m} = 0(m)$

Subsection 4

We know that

original 2nd row R_2 is on P_1

original 3rd row R_3 is on P_2

.
.
.

original mth row R_m is on P_{m-1}

Due to subsection 3 we have

Modified R'_2 on P_1

Modified R'_3 on P_2

.
.
.

Modified R'_m on P_{m-1}

- Add corresponding elements of R'_2 on P_1 and R_2

- Add corresponding elements of R'_3 on P_2 and R_3

.
.
.

- Add corresponding elements R'_m on P_{m-1} and R_m. All these steps can be done in parallel.

• Total time = $2\ m.\alpha = 0\ (m)$

since we have to subtract 2m elements of each row.

5.2 LOAD BALANCING

Load balancing is the issue which comes after implementation. It is required almost in every parallel algorithm implementation. If operating system is not able to manage the data distribution to different processors efficiently, the basic aim of parallel processing will be damaged. In the previous chapter on implementation we assumed good load balancing distributing the load manualy. While implementing on the machine we will use following algorithm which will provide good speed up.

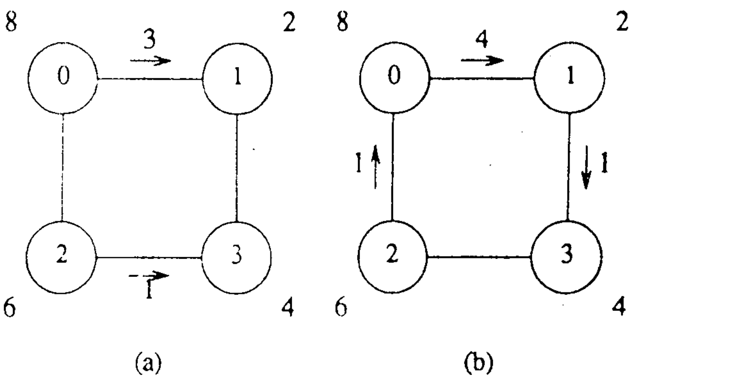
Load balancing used to achieve efficient use of multiprocessor. In the worst case of operating system it can improve the efficiency upto sixty percent.

Suppose we have a hypercube with $p = 2^d$ processors, and each process PE(i) has Li units of load.

Load balancing problem means to redistribute the load so that if L_i is the load on processor i after redistribution then $| L'_i - L'_j | \leq 1$ for every pair of processor i & j.

We are also intrested in minimising the load transfer time.

Example:



We permit several processors to overlap transmission along the same dimension of the hypercube only.

If L_{ij} = load to be tranferred by processor i along dimension j

$0 \leq i < p, \ 0 \leq j < d$

Then load transfer time $T = \sum_j m_j$

where $m_j = \max \{ l_{ij} \}$

Algorithm that minimizes the load transfer time T, when p = 4. When large p, we will use heuristics to minimize T:

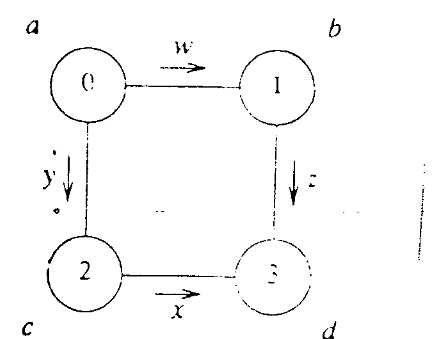


Figure 2: Load redistribution for $p = 4$

Assume $a \geq \max \{b, c, d\}$ let w, x, y and z be such that if thismuch load is transmit in the direction shown, the load is balanced.

Let

$d_0 = w + x$ = total load transfer along dim 0

$d_1 = y + z$ = total load transfer along dim 1

$f = w + y$ = load transfer out of P_0

Solving these equations for w, x & y

$$w = f - d_1 + z$$

$$x = d_0 + d_1 - f - z \qquad (1)$$

$$y = d_1 - z$$

Theorem 1

The load in P_0, P_1, P_2 and P_3 is balanced if d_0, d_1 and f are selected such that

a) $d_0 \frac{a+c-b-d}{2}$ if $a + c \geq b + d$

b) $d_1 \frac{a+b-c-d}{2}$ if $a + b \geq c + d$

c) $f = a - \frac{a+b+c+d}{4}$ if $[(a + c < b + d)$ or $(a + b < c + d)]$ and $(a + b + c + d)\ m_0d_4 = 1$

By using this theorem load can be balanced with min. amount of load enter or leave a processor. It is possible that some w, x, y, z that satisfy theorem 1 do not result in feasible load transfer schemes.

So to ensure feasibility another theorem is required.

Theorem 2

a) if $(-b \leq w \leq a)$ and $(-e \leq x \leq c)$ then the required load transfer can be done by transfering first on dim.0 and then on dim 1.

b) If $(-c \leq y \leq a)$ and $(d \leq z \leq b)$ then we can transfer first on dimension 1 and then on dimension 0.

From eqn. 1

when $w = x$

$$z = \frac{d_0 + 2d_1 - 2f}{2} = \frac{d_0}{2} + d_1 - f$$

At this value of z, $w = x = d_0/2$

Thus from theorems (1) and (2) it follows that T is minimized by selecting z such that:

a) z is an integer that results in a feasible lead transfer w, x, y, z (eqn. 2 and 3).

b) z is an integer in the range

$$\left[\min\left\{\frac{d_o}{2} + d_1 - f,\ \frac{d_1}{2}\right\},\ \max\left\{\frac{d_0}{2} + d_1 - f,\ \frac{d_1}{2}\right\}\right]$$

This contains an integer that satisfies (a). Otherwise z is the closest to this range that result in a feasible load transfer.

c) d_0, d_1 and f are as in Theorem 1. Writing the feasible conditions 2 and 3 using (1):

$$(-b + d_1 - f \leq z \leq a + d_1 - f) \text{ and } (-c + d_0 + d_1 - f \leq z \leq d + d_0 + d_1 - f)$$

or

$$(-a + d_1 \leq z \leq c + d_1) \text{ and } (-d \leq z \leq b) = u \leq z \leq v$$

where

$$u = \max [-b + d_1 - f, -c + d_0 + d_1 - f]$$

$$c = \min [a + d_1 - f, d + d_0 + d_1 - f]$$

or

$$u' \leq z \leq v'$$

where

$$u' = \max [-a + d_1, -d]$$

$$v' = \min [c + d_1, b]$$

Algorithm :

Thus we found following algo to minimize T:

1. Compute d_0, d_1 and f as in Theorem 1.
2. Compute u, v, u′, v′ as above.
3. If there is an integer z in the range [u, v] or [u′, v′] then pick this z and compute w, x, y using (1). If not, then find the integer z nearest to the range [u, v] and [u′, v′] use this z to compute w, x and y.

4. If the selected z is in the range [u, v] then route on dim.0 first otherwise on dim 1 first.

Example

Consider the case of $a = 20$, $b = 4$, $c = 16$, $d = 10$.

From theorem 1 we get

$d_0 = 11$, $d_1 = -1$, $f = 7$

Also $u = -12$, $v = 12$, $u' = -10$, $v' = 4$.

For step 3 we have $[\min \{d_0/2 + d_1 - f, d_1/2\}, \max \{d_0/2 + d_1 - f, d_1/2\}] = [-2.5, -0.5]$

The selected value of $z = -1$.

From (1) we get $w = 7$, $x = 4$ and $y = 0$

The value of T is 8 and we first route on dim.0 and then on dim.1.

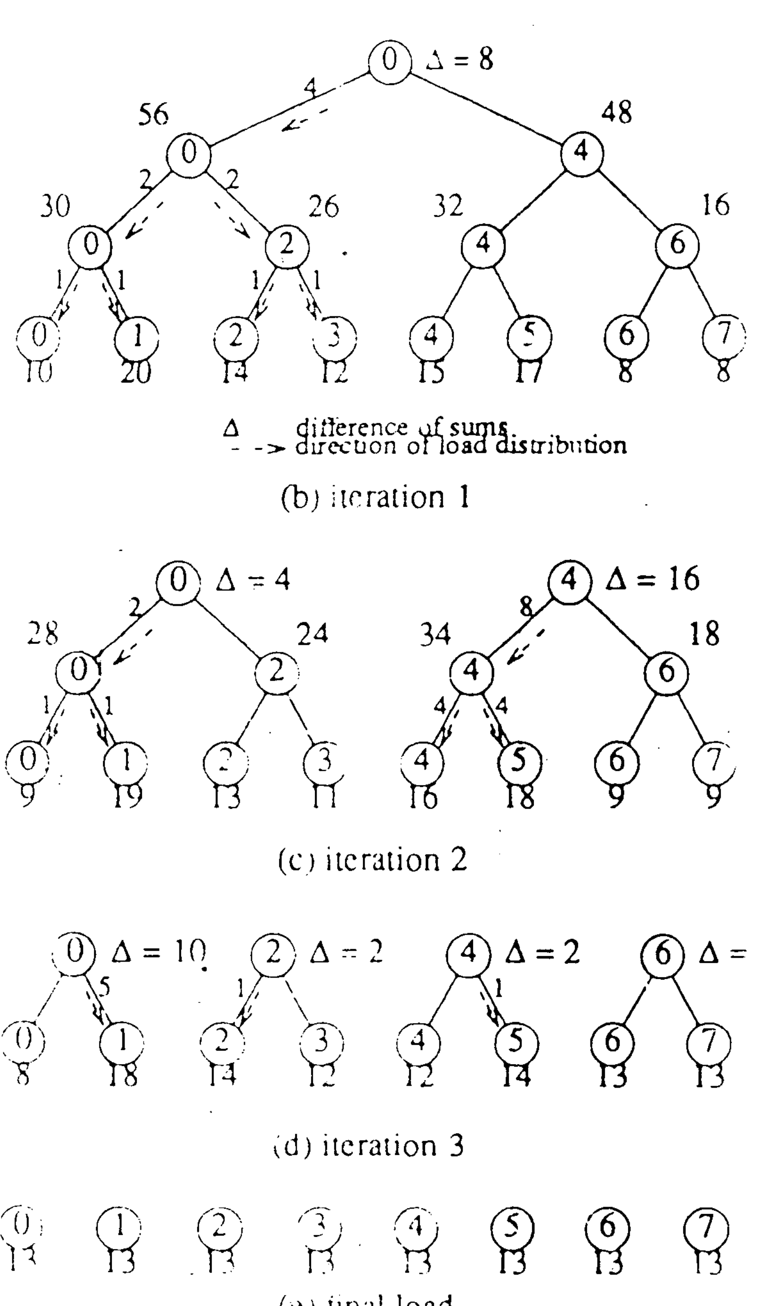
when $p > 4$

<u>Heuristics</u>

Load can be balanced such that $| L'_i - L'_j | \leq 1$ for every pair of process (i) by balancing across each of the d dim of the hypercube in some order.

When balancing across the gth dim. in this order we balance load in pairs of subhypercubes of size 2^{d-v}, $1 \leq q \leq d$.

Consider an 8 processor hypercube with initial load distributions.



(a) initial load

(b) iteration 1

(c) iteration 2

(d) iteration 3

(e) final load

Dimensions considered in the order 2, 1, 0 when considering dim.2 we ensure that total load in each of subhypercube of dim.2 (size 4) differs at most by one.

Algorithm

for r := d-1 downs to 0 d_0

{balancing across dimensions r in pairs of subhypercube of size 2^r}

Perform an upward pens computing sum of loads in the subtree leaves; compute the load difference at each route;

Perform a downward pass to compute load to be transferred;

Transfer the required load;

end;

Time complexity

Since upward and downward pass take 0(r) time, the total time needed

$$= 0\left(d^2 + \sum_{i=0}^{d-1} m_i\right)$$

where m_i = max. load to be transfered between a pair of processor.

Optimization of $\sum_i m_i$

We use a heuristic to determine the processing order for

the dimensions of the hypercube:

The next dimension to balance across is selected by first having each processor i, compute r_i and δ_i such that

δ_i = max { ¦ L_i - L_j ¦ : j is a neighbour of i along an unselected dim. }

r_i is such that δ_i = ¦ L_i - L_j ¦ where j is δ_i's neighbour along dim.r_i

Next max of the si's is computed if this is sw, then dim rw is selected.

The time needed to select the next dim. is 0(d)

Example

For the loads of previous example

s_0 = max { ¦10-20¦, ¦10-14¦, ¦10-15¦} = 10, r_0 = 0

s_1 = max { ¦20-10¦, ¦20-12¦, ¦20-17¦} = 10, r_1 = 0

s_2 = max { ¦14-12¦, ¦14-10¦, ¦14-8¦} = 6, r_2 = 2

s_3 = max { ¦12-14¦, ¦12-20¦, ¦12-8¦} = 8, r_3 = 1

s_4 = max { ¦15-17¦, ¦15-8¦, ¦15-10¦} = 7, r_4 = 1

s_5 = max { ¦17-15¦, ¦17-8¦, ¦17-20¦} = 9, r_5 = 1

s_6 = max { ¦8-8¦, ¦8-15¦, ¦8-14¦} = 7, r_6 = 1

s_7 = max { ¦8-8¦, ¦8-17¦, ¦8-12¦} = 9, r_7 = 1

Since max $\{s_i\} = s_0 = 10$, dim $r_0 = 0$ is selected as the first dim. to balance across. After balancing across this dim. we select from dimensions 1 and 2 the next dim. to balance across. The remaining dim. is balanced across in the last iteration.

The total time spent determining the order of dimensions in $o(d^2)$.

This does not affect the complexity $0(d^2 + \sum_i m_i$ of the load balancing algorithm.

For the example given above and using this heuristic we get

$m_1 = 2,\ m_2 = 4,\ m_3 = 3$

So $\varepsilon\ m_i = 9$

In this heuristic we may require more computations in determining the load transmitted. However, our heuristic can obtain better load balancing and load transfer time.

CHAPTER 6

CONCLUSION

CONCLUSION

Throughout the work we have stressed on three points: efficiency, accuracy & good numerical stability. The conventional algorithms for matrix-inversion are available from a very long time. With the advent of technology, increasing the speeds of logic circuits, memories and input/output equipment and computational capabilities the efficiency of these algorithms has been raised up to a good level. Now the development & research in the field of parallel and distributed completing made possible also to develop very accurate & numerical stable besides efficient parallel algorithms. We have focused on these points in finding the inverse of matrix because this problem can become part of the innerloop of a large calculation, and thus it is essential that it be don efficiently and accurately. This same standard tool can became "buried" in relation to the total problem solution, and it is essential that this tool be reliable. It could be buried so deep that the programmer is not really aware that the tool is being used and an error in it would be very difficult to diagnose.

Almost in every physical system modelling & solving, solving differential equations, solving electrical networks and in much more other problems which need not to be mentioned matrix inverse computations are used. But inverse of matrix has some physical interpretations: There are contain problems in statistics & engineering whose the object of computations

is to see the inverse matrix and not just to use it to find something else. In these problems the elements of A^{-1} have meaning such as being " influence coefficients" that show how forcing terms affect the model.

In our algorithm we have shown two cases of parallelization. In one case we have used m processors and in another case $2m^2$ processors. If we increase the number of processors further more to m^3 we would have got some better results but not upto the expectation of increasing processors by a factor of m/2. In this case our augmented matrix will be of the order of mx(m+1) instead of (mx2m) and we will break the problems into m parts each running simultaneously. Thus if we neglect the communication cost the speed up will be two compared to the previous case pf $2M^2$ processors.

Similarly if we half the number of processor to m^2 and neglect the communication cost the processing time will be increased by a factor of two. In this case we have augmented matrix of the order of (mx2m). We have 2m elements in each row. We have m rows and total number of elements equal of $2m^2$. Thus we will have to do operations in two phases and we take almost twice time as compared to the case of $2m^2$ processors.

From the table given in the chapter of complexity it is clear that factor of "log m" is multiplied when we take communication cost into account. This is due to single node broadcast/Accumulation and multinode broad cost/accumulation

which take log p and p/log p time respectively for one transfer where p is the number of processors.

We have seen that after a limit when we increase number of processors more and more we do not get sufficient speed-up as desired and some-times performance ie degraded due to addition of more processors. This is due to synchronizing and load distribution among processors. As we increase the no of processors the synchronizations as well as communications cost increases and it overdominates computational cost. Also the processing costs for controlling the system and scheduling operations increases drastically after a limit and the basic aim of parallel computing is destroyed.

BIBLIOGRAPHY

1. K.Hwang, F.W, Briggs, " Advanced Computer Architecture and Parallel Processing".
2. Chu,E., George, A.& Grusenel, D. "Parallel Matrix Inversion" Research Report CS-90-48 December 1990.
3. Oscal H. I barra and Myung Hee Kim " Fast Parallel Algorithms for Solving Triangular Systems of Linear Equations on the Hypercube" The Fifth International Parallel Processing Symposium 1991.
4. Khaled M. Elleithy and Magdy A. Bayoumi " from Alogrithms To Parallel Architectures" The Fifth International Parallel Processing Symposium. 1991.
5. Harold S. Stone,"High Performance Computer Architecture".
6. Leach H. Jamieson, Dennis Gangon and Robert J. Douglass. "Characteristics of Parallel Algorithms". MIT Press 1987.
7. T.J. Dekker and W. Hoff mann. " Rehabilitation of the Gauss-Jordan algorithm" Journal Numer. Mathematik 591-599, 1989.
8. Lydia Kronsjo, "Algorithms: Their Complexity and Efficiency".
9. Marios M. Polycarpou and Petros A. Ioannou " Parallel Algorithm for Fast. Matrix Inversion" The Fifth International Parallel Processing Symposium. 1991.
10. John R. Rice " Matrix Compulations And Mathematical Software".
11. S.L. Johnson. " Communications effecient basic linear algebra computations on hypercube architectures" J. Parallel Distrib. comput. 133-172, 1987.

12. W.D. Hillis and G.L Steel. "Data Parallel Algorithms". Comm.ACM, December 1986, pages 1170-1183.

13. Thesis, D.J., Titus, Harold, " Parallel Processor Systems, Technologies and Applications".

14. Jinwoon Woo and Sartaj Sahni " Load Balancing on a Hypercube". The Fifth International Symposium on Parallel Processing 1991 IEEE.

15. Branko Soucek and Mariana Soucek " Neural and Massively Parallel Computers".

16. Dimitri P. Brestsekas and John N. Tsitskilis, " Parallel And Distributed Computation: Numerical Methods".