UNIT-VI

NP Hard and NP complete problems: Nondeterministic algorithms, The classes NP hard and NP complete; NP hard graph problems - Clique decision problem (CDP).

PRAM Algorithms: Introduction, Computational Model.

NP-Hard and NP-Complete problems

Deterministic and non-deterministic algorithms

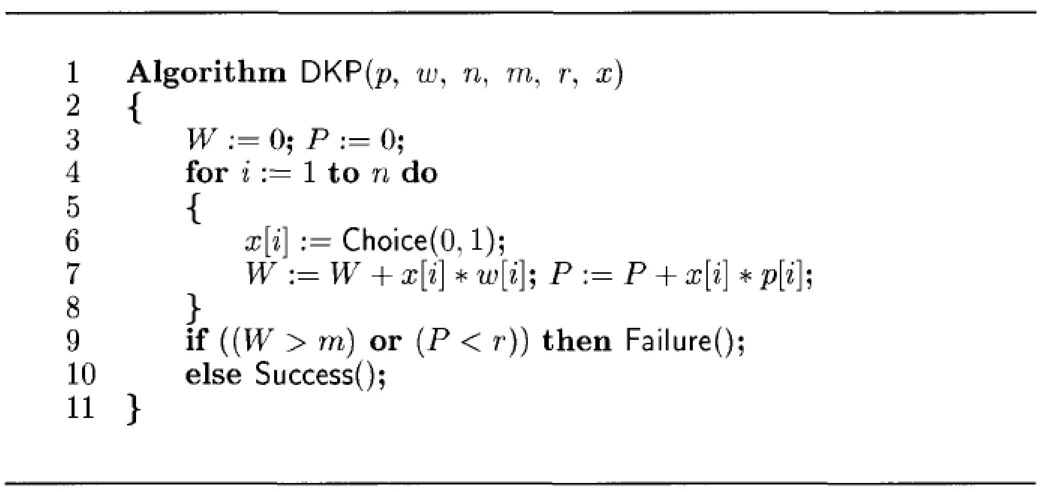
Deterministic: The algorithm in which every operation is uniquely defined is called deterministic algorithm.
Non-Deterministic: The algorithm in which the operations are not uniquely defined but are limited to specific set of possibilities for every operation, such an algorithm is called non-deterministic algorithm.
The non-deterministic algorithms use the following functions:

1. Choice: Arbitrarily chooses one of the element from given set.
2. Failure: Indicates an unsuccessful completion
3. Success: Indicates a successful completion

A non-deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal. Whenever, there is a set of choices that leads to a successful completion, then one such set of choices is selected and the algorithm terminates successfully.
In case the successful completion is not possible, then the complexity is O(1). In case of successful signal completion then the time required is the minimum number of steps needed to reach a successful completion of O(n) where n is the number of inputs.
The problems that are solved in polynomial time are called tractable problems and the problems that require super polynomial time are called non-tractable problems. All deterministic polynomial time algorithms are tractable and the non-deterministic polynomials are intractable.

```
Algorithm NSort(A, n)
// Sort n positive integers.
{
    for i := 1 to n do B[i] := 0; // Initialize B[ ].
    for i := 1 to n do
    {
        j := Choice(1, n);
        if B[j] ≠ 0 then Failure();
        B[j] := A[i];
    }
    for i := 1 to n − 1 do  // Verify order.
        if B[i] > B[i + 1] then Failure();
    write (B[1 : n]);
    Success();
}
```

Nondeterministic sorting



```
Algorithm DKP($p$, $w$, $n$, $m$, $r$, $x$)
{
    $W := 0$; $P := 0$;
    for $i := 1$ to $n$ do
    {
        $x[i] :=$ Choice(0, 1);
        $W := W + x[i] * w[i]$; $P := P + x[i] * p[i]$;
    }
    if (($W > m$) or ($P < r$)) then Failure();
    else Success();
}
```

Nondeterministic knapsack algorithm

Satisfiability Problem:

The satisfiability is a boolean formula that can be constructed using the following literals and operations.

1. A literal is either a variable or its negation of the variable.
2. The literals are connected with operators $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$
3. Parenthesis

The satisfiability problem is to determine whether a Boolean formula is true for some assignment of truth values to the variables. In general, formulas are expressed in Conjunctive Normal Form (CNF).

A Boolean formula is in conjunctive normal form iff it is represented by
$(x_i \vee x_j \vee x_k^1)$ A $(x_i \vee x^1_j \vee x_k)$

A Boolean formula is in 3CNF if each clause has exactly 3 distinct literals.

Example:

The non-deterministic algorithm that terminates successfully iff a given formula E(x1,x2,x3) is satisfiable.

```
Algorithm Eval($E$, $n$)
// Determine whether the propositional formula $E$ is
// satisfiable. The variables are $x_1, x_2, \ldots, x_n$.
{
    for $i := 1$ to $n$ do  // Choose a truth value assignment.
        $x_i$ := Choice(false, true);
    if $E(x_1, \ldots, x_n)$ then Success();
    else Failure();
}
```

Nondeterministic satisfiability

Reducability:
A problem Q1 can be reduced to Q2 if any instance of Q1 can be easily rephrased as an instance of Q2. If the solution to the problem Q2 provides a solution to the problem Q1, then these are said to be reducable problems.

Let L1 and L2 are the two problems. L1 is reduced to L2 iff there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time and is denoted by L1α L2.
If we have a polynomial time algorithm for L2 then we can solve L1 in polynomial time.
Two problems L1 and L2 are said to be polynomially equivalent iff L1α L2 and L2 α L1.

Example: Let P1 be the problem of selection and P2 be the problem of sorting. Let the input have n numbers. If the numbers are sorted in array A[ ] the i^{th} smallest element of the input can be obtained as A[i]. Thus P1 reduces to P2 in O(1) time.

Decision Problem:
Any problem for which the answer is either yes or no is called decision problem. The algorithm for decision problem is called decision algorithm.
Example: Max clique problem, sum of subsets problem.

Optimization Problem: Any problem that involves the identification of an optimal value (maximum or minimum) is called optimization problem.
Example: Knapsack problem, travelling salesperson problem.
In decision problem, the output statement is implicit and no explicit statements are permitted.
The output from a decision problem is uniquely defined by the input parameters and algorithm specification.

Many optimization problems can be reduced by decision problems with the property that a decision problem can be solved in polynomial time iff the corresponding optimization problem can be solved in polynomial time. If the decision problem cannot be solved in polynomial time then the optimization problem cannot be solved in polynomial time.

Class *P*:

<u>*P*</u>: the class of decision problems that are solvable in O($p(n)$) time, where $p(n)$ is a polynomial of problem's input size n

Examples:

- searching
- element uniqueness
- graph connectivity
- graph acyclicity
- primality testing

Class *NP*

<u>*NP*</u> (<u>*nondeterministic polynomial*</u>): class of decision problems whose proposed solutions can be verified in polynomial time = solvable by a *nondeterministic polynomial algorithm*

A <u>*nondeterministic polynomial algorithm*</u> is an abstract two-stage procedure that:

- generates a random string purported to solve the problem
- checks whether this solution is correct in polynomial time

By definition, it solves the problem if it's capable of generating and verifying a solution on one of its tries

Example: CNF satisfiability

Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there values of its variables that makes it true? This problem is in *NP*.

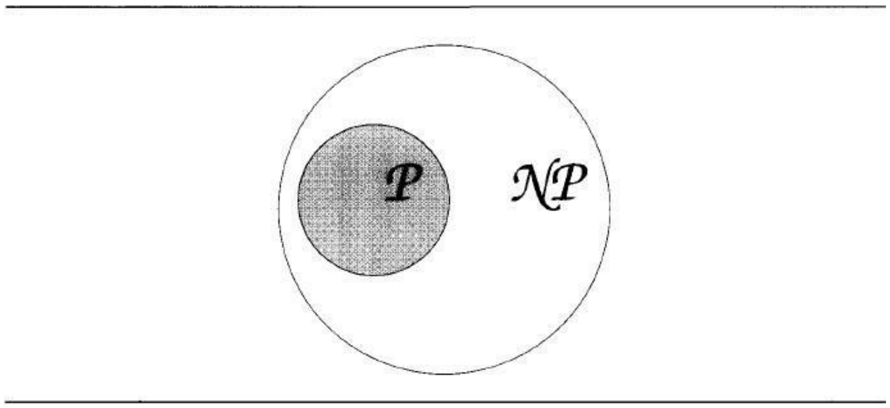
Nondeterministic algorithm:

- Guess truth assignment
- Substitute the values into the CNF formula to see if it evaluates to true

What problems are in *NP*?

- Hamiltonian circuit existence
- Partition problem: Is it possible to partition a set of n integers into two disjoint subsets with the same sum?
- Decision versions of TSP, knapsack problem, graph coloring, and many other combinatorial optimization problems. (Few exceptions include: MST, shortest paths)
- All the problems in *P* can also be solved in this manner (but no guessing is necessary), so we have:

  $P \subseteq NP$
- Big question: $P = NP$?



Commonly believed relationship between $\mathcal{P}$ and $\mathcal{NP}$

NP HARD AND NP COMPLETE

Polynomial Time algorithms

Problems whose solutions times are bounded by polynomials of small degree are called polynomial time algorithms

Example: Linear search, quick sort, all pairs shortest path etc.

Non- Polynomial time algorithms

Problems whose solutions times are bounded by non-polynomials are called non-polynomial time algorithms

Examples: Travelling salesman problem, 0/1 knapsack problem etc

It is impossible to develop the algorithms whose time complexity is polynomial for non-polynomial time problems, because the computing times of non-polynomial are greater than polynomial. A problem that can be solved in polynomial time in one model can also be solved in polynomial time.

NP-Hard and NP-Complete Problem:

Let P denote the set of all decision problems solvable by deterministic algorithm in polynomial time. NP denotes set of decision problems solvable by nondeterministic algorithms in polynomial time. Since, deterministic algorithms are a special case of nondeterministic algorithms, P ⊆ NP. The nondeterministic polynomial time problems can be classified into two classes. They are

1. NP Hard and
2. NP Complete

NP-Hard: A problem L is NP-Hard iff satisfiability reduces to L i.e., any nondeterministic polynomial time problem is satisfiable and reducable then the problem is said to be NP-Hard.

Example: Halting Problem, Flow shop scheduling problem

NP-Complete: A problem L is NP-Complete iff L is NP-Hard and L belongs to NP (nondeterministic polynomial).

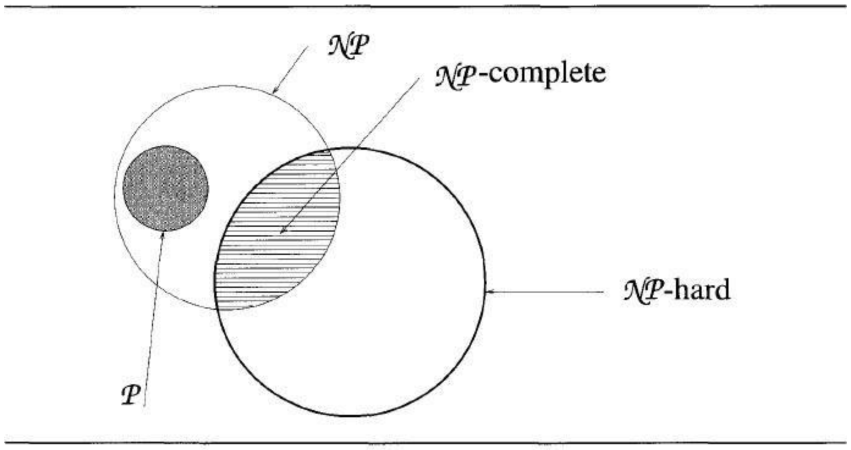
A problem that is NP-Complete has the property that it can be solved in polynomial time iff all other NP-Complete problems can also be solved in polynomial time. (NP=P)

If an NP-hard problem can be solved in polynomial time, then all NP- complete problems can be solved in polynomial time. All NP-Complete problems are NP-hard, but some NP-hard problems are not known to be NP- Complete.
Normally the decision problems are NP-complete but the optimization problems are NP-Hard.

However if problem L1 is a decision problem and L2 is an optimization problem, then it is possible that L1α L2.
Example: Knapsack decision problem can be reduced to knapsack optimization problem.
There are some NP-hard problems that are not NP-Complete.

Relationship between P,NP,NP-hard, NP-Complete

Let P, NP, NP-hard, NP-Complete are the sets of all possible decision problems that are solvable in polynomial time by using deterministic algorithms, non-deterministic algorithms, NP-Hard and NP-complete respectively. Then the relationship between P, NP, NP-hard, NP-Complete can be expressed using Venn diagram as:



Commonly believed relationship among $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard problems

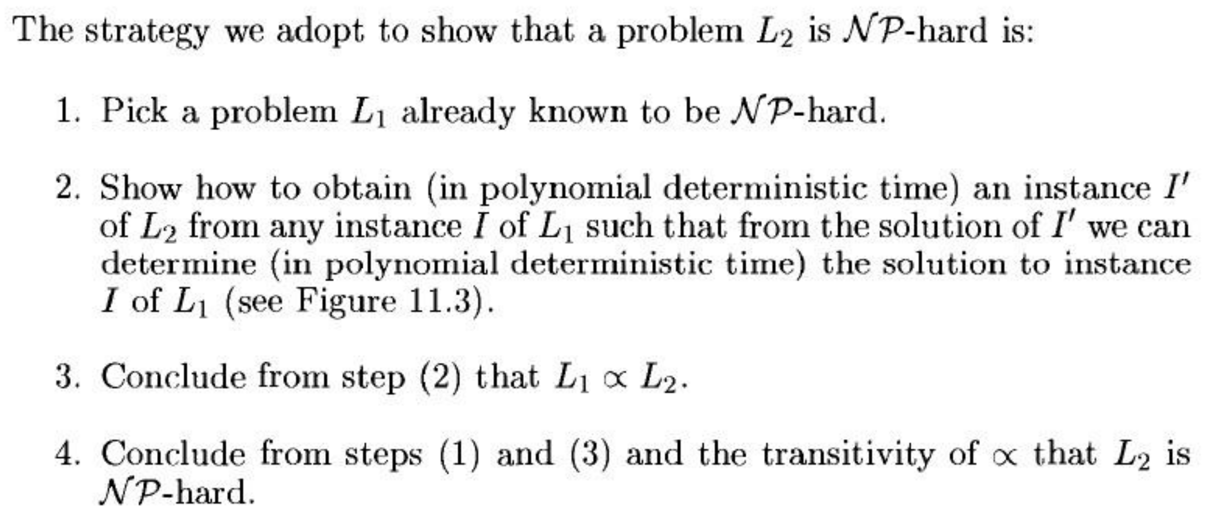
Problem conversion

A decision problem D1 can be converted into a decision problem D2 if there is an algorithm which takes as input an arbitrary instance I1 of D1 and delivers as output an instance I2 of D2such that I2 is a positive instance of D2 if and only if I1 is a positive instance of D1. If D1 can be converted into D2, and we have an algorithm which solves D2, then we thereby have an algorithm which solves D1. To solve an instance I of D1, we first use the conversion algorithm to generate an instance I0 of D2, and then use the algorithm for solving D2 to determine whether or not I0 is a positive instance of D2. If it is, then we know that I is a positive instance of D1, and if it is not, then we know that I is a negative instance of D1. Either way, we have solved D1 for that instance. Moreover, in this case, we can say that the computational complexity of D1 is at most the sum of the computational complexities of D2 and the conversion algorithm. If the conversion algorithm has polynomial complexity, we say that D1 is at most polynomially harder than D2. It means that the amount of computational work we have to do to solve D1, over and above whatever is required to solve D2, is polynomial in the size of the problem instance. In such a case the conversion algorithm provides us with a feasible way of solving D1, given that we know how to solve D2.

Given a problem X, prove it is in NP-Complete.

1. Prove X is in NP.
2. Select problem Y that is known to be in NP-Complete.
3. Define a polynomial time reduction from Y to X.
4. Prove that given an instance of Y, Y has a solution iff X has a solution.

NP-Hard Graph Problems



The strategy we adopt to show that a problem L_2 is $\mathcal{NP}$-hard is:

1. Pick a problem L_1 already known to be $\mathcal{NP}$-hard.
2. Show how to obtain (in polynomial deterministic time) an instance I' of L_2 from any instance I of L_1 such that from the solution of I' we can determine (in polynomial deterministic time) the solution to instance I of L_1 (see Figure 11.3).
3. Conclude from step (2) that $L_1 \propto L_2$.
4. Conclude from steps (1) and (3) and the transitivity of $\propto$ that L_2 is $\mathcal{NP}$-hard.

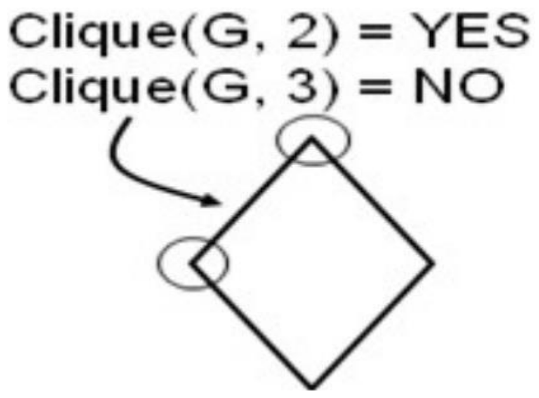
Clique Decision Problem

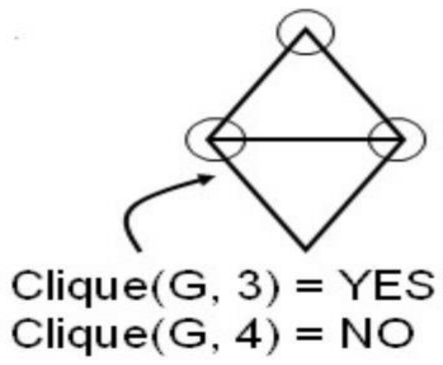
Definition: - It is a sub graph of the given graph which is complete.

In Clique, every vertex is directly connected to another vertex, and the number of vertices in the Clique represents the Size of Clique.

Maximal Clique:

Clique is a maximal complete sub-graph of a graph G = (V,E), that is a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph).





The Size of a clique is the number of vertices in it. The Maximal clique problem is an optimization problem that has to determine the size of a largest clique in G. A decision problem is to determine whether G has a clique of size at least ‘k’.

Example 11.11 Consider $F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$. The construction of Theorem 11.2 yields the graph of Figure 11.4. This graph contains six cliques of size two. Consider the clique with vertices $\{\langle x_1, 1\rangle, \langle \bar{x}_2, 2\rangle\}$. By setting x_1 = true and $\bar{x}_2$ = true (that is, x_2 = false), F is satisfied. The x_3 may be set either to true or false. □

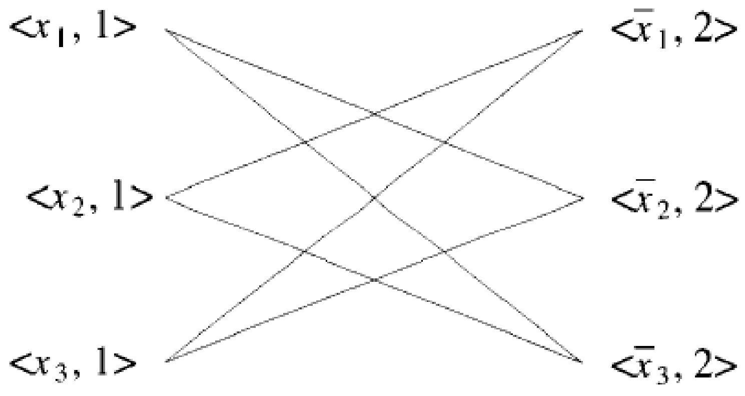


Figure 11.4 A sample graph and satisfiability

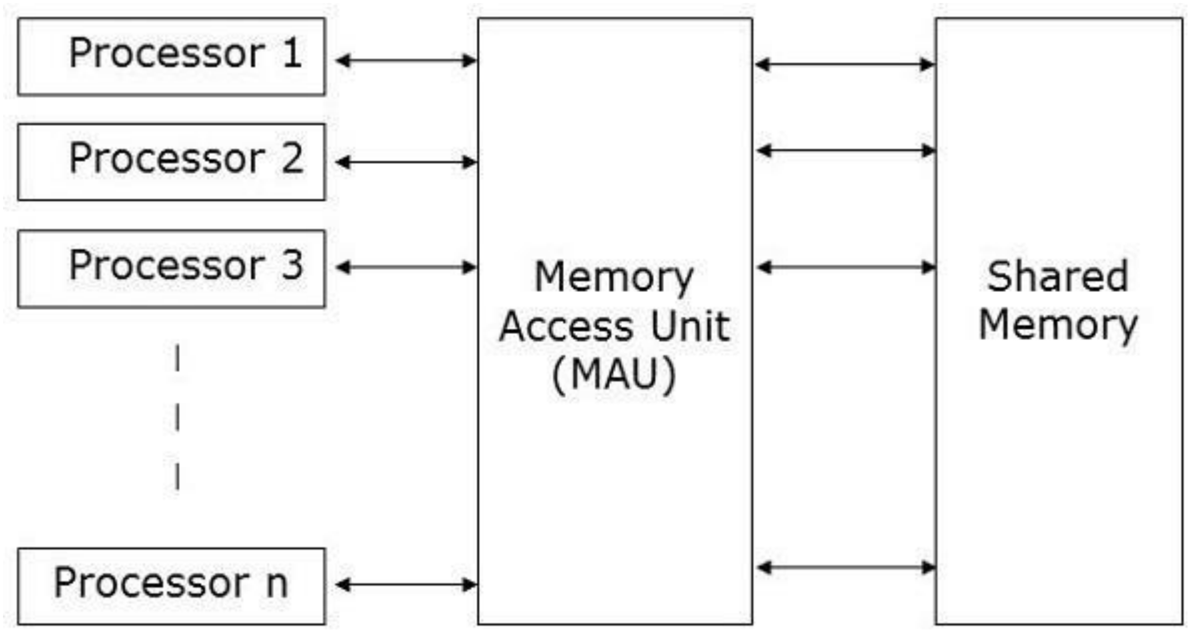
Parallel Random Access Machines (PRAM)

Parallel Random Access Machines (PRAM) is a model, which is considered for most of the parallel algorithms. Here, multiple processors are attached to a single block of memory. A PRAM model contains −

A set of similar type of processors.

All the processors share a common memory unit. Processors can communicate among themselves through the shared memory only.

A memory access unit (MAU) connects the processors with the single shared memory.



Here, **n** number of processors can perform independent operations on **n** number of data in a particular unit of time. This may result in simultaneous access of same memory location by different processors.

To solve this problem, the following constraints have been enforced on PRAM model −

Exclusive Read Exclusive Write (EREW) − Here no two processors are allowed to read from or write to the same memory location at the same time.

Exclusive Read Concurrent Write (ERCW) − Here no two processors are allowed to read from the same memory location at the same time, but are allowed to write to the same memory location at the same time**oncurrent Read Exclusive Write (CREW)** − Here all the processors are allowed to read from the same memory location at the same time, but are not allowed to write to the same memory location at the same time.
Concurrent Read Concurrent Write (CRCW) − All the processors are allowed to read from or write to the same memory location at the same time.

-

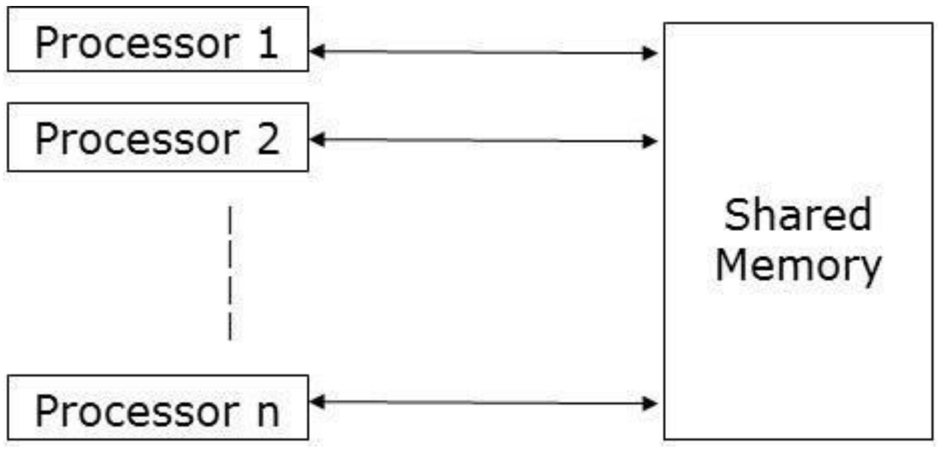
There are many methods to implement the PRAM model, but the most prominent ones are −

- Shared memory model
- Message passing model
- Data parallel model

Shared Memory Model

Shared memory emphasizes on **control parallelism** than on **data parallelism**. In the shared memory model, multiple processes execute on different processors independently, but they share a common memory space. Due to any processor activity, if there is any change in any memory location, it is visible to the rest of the processors.

As multiple processors access the same memory location, it may happen that at any particular point of time, more than one processor is accessing the same memory location. Suppose one is reading that location and the other is writing on that location. It may create confusion. To avoid this, some control mechanism, like **lock / semaphore,** is implemented to ensure mutual exclusion.



Shared memory programming has been implemented in the following −

-
Thread libraries − The thread library allows multiple threads of control that run concurrently in the same memory location. Thread library provides an interface that supports multithreading through a library of subroutine. It contains subroutines for

- Creating and destroying threads
- Scheduling execution of thread

- passing data and message between threads
- saving and restoring thread contexts
-

Examples of thread libraries include − SolarisTM threads for Solaris, POSIX threads as implemented in Linux, Win32 threads available in Windows NT and Windows 2000, and JavaTM threads as part of the standard JavaTM Development Kit (JDK).

-

Distributed Shared Memory (DSM) Systems − DSM systems create an abstraction of shared memory on loosely coupled architecture in order to implement shared memory programming without hardware support. They implement standard libraries and use the advanced user-level memory management features present in modern operating systems. Examples include Tread Marks System, Munin, IVY, Shasta, Brazos, and Cashmere.

-
-

Program Annotation Packages − This is implemented on the architectures having uniform memory access characteristics. The most notable example of program annotation packages is OpenMP. OpenMP implements functional parallelism. It mainly focuses on parallelization of loops.

-

The concept of shared memory provides a low-level control of shared memory system, but it tends to be tedious and erroneous. It is more applicable for system programming than application programming.

Merits of Shared Memory Programming

Global address space gives a user-friendly programming approach to memory.

Due to the closeness of memory to CPU, data sharing among processes is fast and uniform.

There is no need to specify distinctly the communication of data among processes.

Process-communication overhead is negligible.

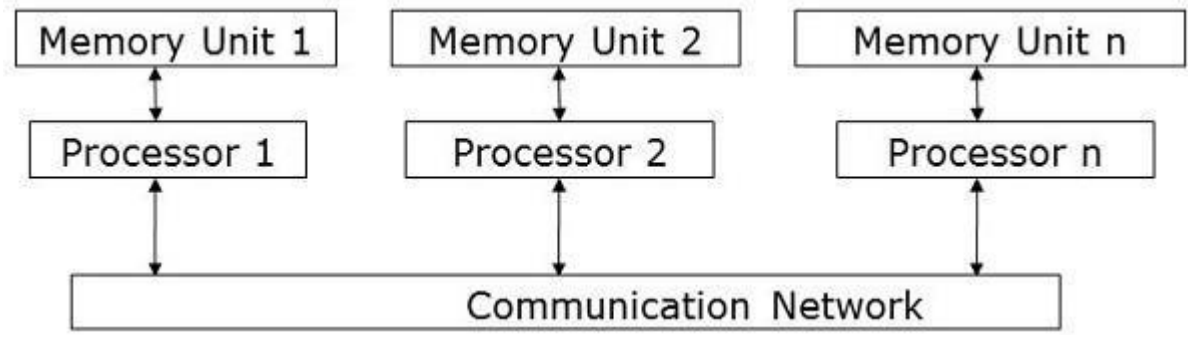
It is very easy to learn.

Demerits of Shared Memory Programming

- It is not portable.
- Managing data locality is very difficult.

Message Passing Model

Message passing is the most commonly used parallel programming approach in distributed memory systems. Here, the programmer has to determine the parallelism. In this model, all the processors have their own local memory unit and they exchange data through a communication network.



Processors use message-passing libraries for communication among themselves. Along with the data being sent, the message contains the following components −

The address of the processor from which the message is being sent;

Starting address of the memory location of the data in the sending processor;

Data type of the sending data;

Data size of the sending data;

The address of the processor to which the message is being sent;

Starting address of the memory location for the data in the receiving processor.

Processors can communicate with each other by any of the following methods −

- Point-to-Point Communication
- Collective Communication
- Message Passing Interface

Point-to-Point Communication

Point-to-point communication is the simplest form of message passing. Here, a message can be sent from the sending processor to a receiving processor by any of the following transfer modes −

Synchronous mode − The next message is sent only after the receiving a confirmation that its previous message has been delivered, to maintain the sequence of the message.

Asynchronous mode − To send the next message, receipt of the confirmation of the delivery of the previous message is not required.

Merits of Message Passing

- Provides low-level control of parallelism;
- It is portable;
- Less error prone;
- Less overhead in parallel synchronization and data distribution.

Demerits of Message Passing

As compared to parallel shared-memory code, message-passing code generally needs more software overhead.

-

- Message Passing Interface (MPI)
- Parallel Virtual Machine (PVM)

Message Passing Interface (MPI)

It is a universal standard to provide communication among all the concurrent processes in a distributed memory system. Most of the commonly used parallel computing platforms provide at least one implementation of message passing interface. It has been implemented as the collection of predefined functions called **library** and can be called from languages such as C, C++, Fortran, etc. MPIs are both fast and portable as compared to the other message passing libraries.

Merits of Message Passing Interface

Runs only on shared memory architectures or distributed memory architectures;

Each processors has its own local variables;

As compared to large shared memory computers, distributed memory computers are less expensive.

Demerits of Message Passing Interface

- More programming changes are required for parallel algorithm;
- Sometimes difficult to debug; and
- Does not perform well in the communication network between the nodes.

Parallel Virtual Machine (PVM)

PVM is a portable message passing system, designed to connect separate heterogeneous host machines to form a single virtual machine. It is a single manageable parallel computing resource. Large computational problems like superconductivity studies, molecular dynamics simulations, and matrix algorithms can be solved more cost effectively by using the memory and the aggregate power of many computers. It manages all message routing, data conversion, task scheduling in the network of incompatible computer architectures.

Features of PVM

- Very easy to install and configure;
- Multiple users can use PVM at the same time;
- One user can execute multiple applications;
- It's a small package;
- Supports C, C++, Fortran;
- For a given run of a PVM program, users can select the group of machines;
- It is a message-passing model,

- Process-based computation;
- Supports heterogeneous architecture.
- Arrays
- Hypercube Network

Hypercube Network

Hypercube architecture is helpful for those parallel algorithms where each task has to communicate with other tasks. Hypercube topology can easily embed other topologies such as ring and mesh. It is also known as n-cubes, where **n** is the number of dimensions. A hypercube can be constructed recursively.

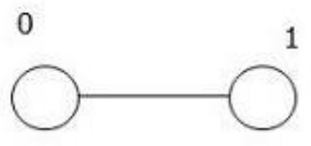


Figure (a) 1-cube

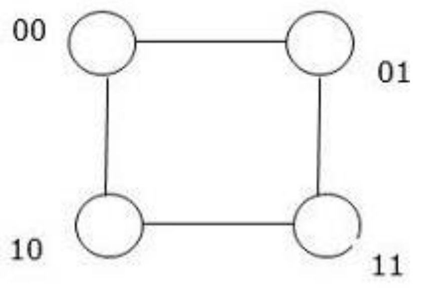


Figure (b) 2-cube

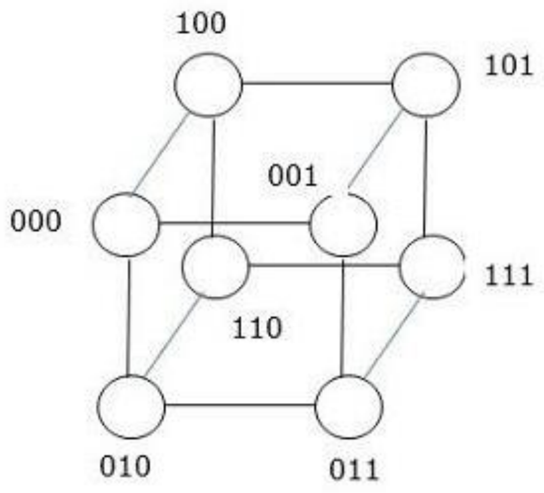


Figure (c) 3-cube

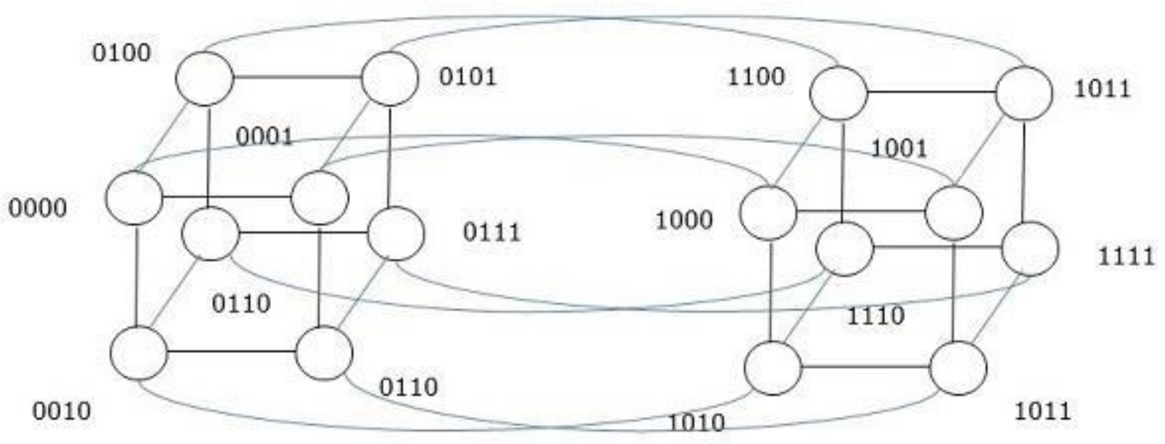


Figure (d) 4-cube