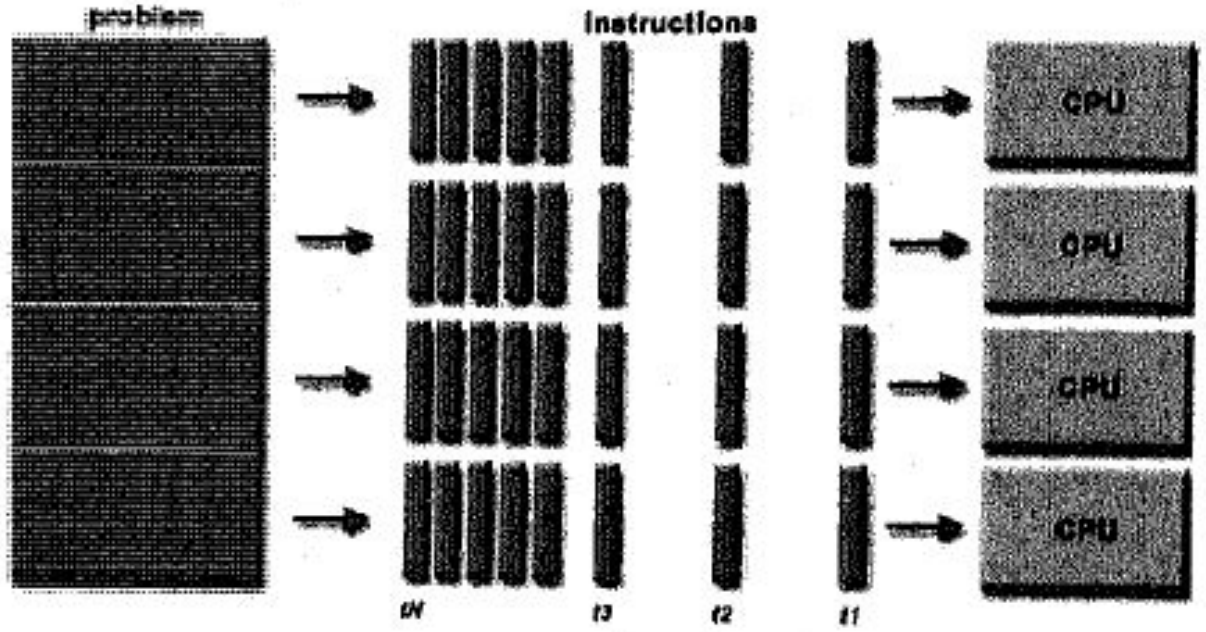
PARALLEL AND DISTRIBUTED

ANU YADAV

PARALLEL COMPUTING

In the simplest sense, *parallel computing* is the simultaneous use of multiple compute resources to solve a computational problem.

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



The compute resources can include :

- A single computer with multiple processors ;
- An arbitrary number of computers connected by a network ;
- A combination of both.

The computational problem usually demonstrates characteristics such as the ability to be :

- Broken apart into discrete pieces of work that can be solved simultaneously ;
- Execute multiple program instructions at any moment in time ;
- Solved in less time with multiple compute resources than with a single compute resource.

Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world : many complex, interrelated events happening at the same time, yet within a sequence.

EXAMPLE

- Weather patterns
- Oceans patterns
- Daily operations within a business
- Building of shopping mall
- Automobile assembly
- Planetary orbits

WHY USE PARALLEL COMPUTING?

- Save time
- Solve larger problem
- Provide concurrency
- Others: cost saving

Limits to serial computing both physical and practical reasons pose significant constraints to simply building ever faster serial computers :

- *Transmission speeds*. the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
- *Limits to miniaturization*. processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
- *Economic limitations*. It is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

VON NEUMANN ARCHITECTURE

- MEMORY IS USED TO STORE BOTH PROGRAM AND DATA INSTRUCTIONS
- Data is information to be used by the program.
- DATA TYPE: Nominal (category: gender, marital status), Ordinal (category, order: ranking of college), Interval (temp, IQ, Gate Score/CAT score), ratio (BMI, Height, weight, Age)
- Program instructions are coded data which tell the computer to do something
- A CPU gets instructions and data from memory , decodes the instructions and then sequentially performs them.
- Math: John Von Neumann, it is also called as machine model.
- It uses stored program concept
- The cpu executes a stored program that specifies a sequence of read and write operations on memory

VON NEUMANN ARCH

FLYNN'S CLASSICAL TAXONOMY

- In parallel computers, there are 4 different possible matrix classification, it is also called as Flynn taxonomy. (from 1966). Two independent dimensions are Data and Instruction. Two possible states are : single or multiple
- SISD: SINGLE Instruction, Single Data
- SIMD: SINGLE Instruction, Multiple Data
- MISD: Multiple Instruction, Single Data
- MIMD: Multiple Instruction, Multiple Data

SISD

- Single instructions means only one instruction stream is being acted on by the CPU during any One clock cycle.
- Single data means only one data stream is being used as input during one clock cycle.

SIMD

- Single instruction : all processing units execute the same instruction at any given clock cycle.
- Multiple data : each processing unit can operate on a different data element.
- Example image processing
- Two varieties : Processor array, vector pipelines

MISD

Multiple instruction and single data: a single data stream is fed into multiple processing units.

Each processing units operates on the data independently via independent instructions stream.

MIMD

- Most commonly used parallel computer
- Example supercomputers, mutli processor SMP computers
- Multiple instructions: every processor execute a different instruction streams
- Multiple data: Every processor working with different data stream.

PARALLEL ALGORITHM

- It is the algorithm that perform more than one operation at a time. RAM will perform parallel computing not serial/seq.
- Parallel algo in list, tree, graph, arrays.
- The processor (0,1, 2,… p-1) serial, shared global memory. All processor can read from and write to the global memory "in parallel". The processor can perform various arithmetic and logical operations in parallel
- Key assumption: PRAM model is that running time can be measured number of parallel memory accessed – algorithm performed. The running time for parallel algorithm depends on the number of processor executing the algorithm as well as the size of the problem input.
- Time and processor count

CONCURRENT VS EXCLUSION MEMORY ACCESS

- Concurrent read algorithm is a PRAM Algorithm during execution of multiple processor can read from same location of shared memory
- Exclusive read algorithm is a PRAM algorithm in which no two processor ever read the same memory location at the same time.
- A pram that support only EREW Algorithm it is called as EREW PRAM
- A pram that support only CRCW Algorithm it is called as CRCW PRAM
- A pram that support only CREW Algorithm it is called as CREW PRAM
- A pram that support only ERCW Algorithm it is called as ERCW PRAM

LIST RANKING BY POINTER JUMPING

32.7 List Ranking by Pointer Jumping

We can store a list in a PRAM much as we store lists in an ordinary RAM. To operate on list objects in parallel, however, it is convenient to assign a "responsible" processor to each object. We shall assume that there are as many processors as list objects, and that the ith processor is responsible for the ith object. For example, shows a linked list consisting of the sequence of objects $\langle 2,4,6,1,8,5\rangle$. Since there is one processor per list object, every object in the list can be operated on by its responsible processor in $O(1)$ time.

An efficient parallel solution, requiring only $O(\lg n)$ time, is given by the following pseudo code.

```
LIST-RANK (L)
1.  for each processor i, in parallel
2.      do if next[i] = NIL
3.          then d [i] ← 0
4.          else d [i] ← 1
5.  while there exists an object i such that next[i] ≠ NIL
6.      do for each processor i, in parallel
7.          do if next[i] ≠ NIL
8.              then d [i] ← d [i] + d[next[i]]
9.                   next[i] ← next[next[i]]
```

The idea implemented by line 9, in which we set next[i] ← next[next[i]] for all non-nil pointers next[i], is called **pointer jumping**. Note that the pointer fields are changed by pointer jumping, thus destroying the structure of the list. If the list structure must be preserved, then we make copies of the next pointers and use the copies to compute the distances.

Parallel prefix on a list

A prefix computation is defined in terms of a binary, associative operator $\otimes$. The computation takes as input a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ and produces as output a sequence $\langle y_1, y_2, \ldots, y_n \rangle$ such that $y_1 = x_1$ and

$$y_k = y_{k-1} \otimes x_k$$
$$= x_1 \otimes x_2 \otimes \ldots \otimes x_k$$

For $k = 2, 3, \ldots, n$. In other words, each y_k is obtained by "multiplying" together the first k elements of the sequence of x_k-hence, the term "prefix."

```
LIST-PREFIX(L)
1.  for each processor i, in parallel
2.      do y[i] ← x[i]
3.  while there exists an object i such that next[i] ≠ NIL
4.      do for each processor i, in parallel
5.          do if next[i] ≠ NIL
6.              then y[next[i]] ← y[i] ⊗ y[next[i]]
7.                   next[i] ← next[next[i]]
```

In List-Rank, processor i updates $d[i]$ - its own d value - whereas in LIST-PREFIX, processor i updates $y[\text{next}[i]]$ - another processor's y value.

APPROXIMATE ALGORITHM

- An approximate algorithm is a way of dealing with NP COMPLETEness for optimization problem. The goal of approximation algorithm is find optimal solution (closer to your exact solution) in a polynomial time.
- C : cost of solution
- C*: cost of optimal solution
- P(n): approximation Ratio
- Maximization: (C*/ C) <= P(n)
- Minimization: (C/ C*) <= P(n)
- P(n) < 1 (not less than 1)
- n : input size

APPROXIMATE ALGORITHM

- It is also called as heuristic algorithms.
- They are used to solve many of approaches, one of example NP Complete Optimization problem.
- NPC to solve this problem, we have three things/ methods to solve NPC.
  - Time Algorithm : we find the optimal solution but that is not feasible if problem size is large.
  - Optimization methods: Branch and bound, genetic algorithms, neural nets/ networks. It is a hard / complex due to more mathematics to show how good they are as compared to the other solution.
  - Approximate algorithms: Generally, they are very fast , optimal solution
- Approximation solution: A feasible solution with value close to the value of optimal solution, it is called as Approximation solution. An approximate algorithm for P is an algorithm that generates approximate solution for P.

- Goal Of an approximation to come as close as to the optimal value as possible in reasonable amount of time

For example, suppose we are looking for a **minimum size vertex-cover** (VC). An approximate algorithm returns a VC for us, but the size (cost) may not be minimum.

Another example is we are looking for a **maximum size independent set** (IS). An approximate algorithm returns an IS for us, but the size (cost) may not be Maximum. Let C be the cost of the solution returned by an approximate algorithm, and C^* is the cost of the optimal solution.

We say the approximate algorithm has an *approximation ratio* $P(n)$ for an input size n, where

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq P(n)$$

Intuitively, the *approximation ratio* measures how bad the approximate solution is compared with the optimal solution. A large (small) *approximation ratio* means the solution is much worse than (more or less the same as) an optimal solution.

Observe that $P(n)$ is always ≥ 1 ; if the ratio does not depend on n, we may just write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have

Imagine we are a salesman, and we need to visit n cities. We want to start a *tour* at a city and visit every city *exactly one time*, and finish the tour at the city from where we start. There is a non-negative cost $c(i, j)$ to travel from city i to city j. The goal is to find a tour (which is a Hamiltonian cycle) of minimum cost. We assume every two cities are connected. Such problem is called *Traveling-salesman problem* (TSP).

We can model the cities as a complete graph of n vertices, where each vertex represents a city.

It can be shown that TSP is NPC.

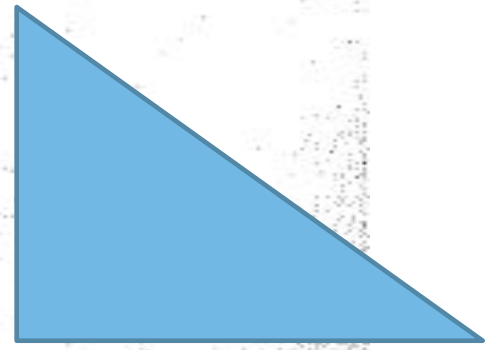
If we assume the cost function c satisfies the *triangle inequality*, then we can use the following approximate algorithm.

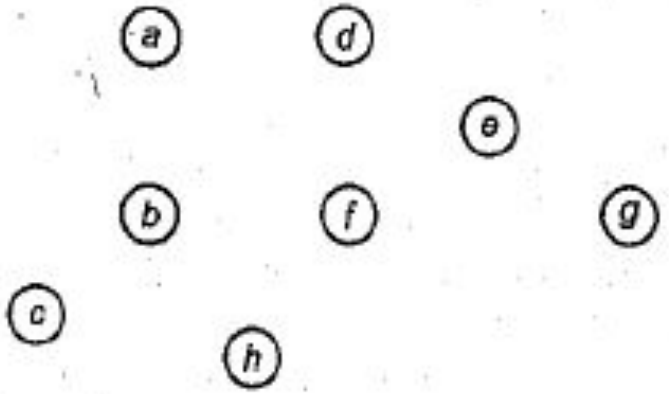
Triangle inequality. Let u, v, w be any three vertices, we have

$$c(u, w) \leq c(u, v) + c(v, w)$$

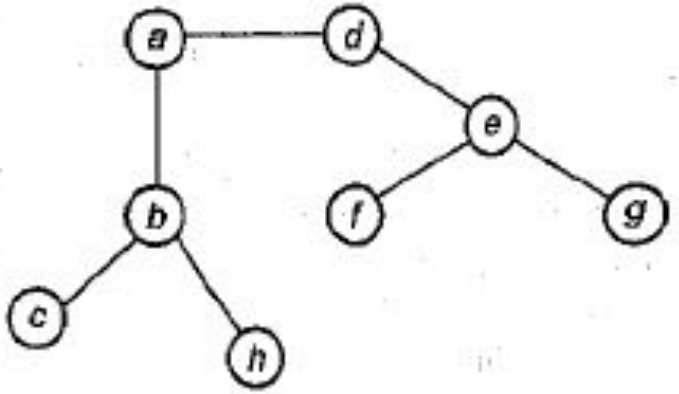
One important observation to develop an approximate solution is if we remove an edge from H^*, the tour becomes a *spanning tree*.

```
Approx-TSP (G = (V, E)) {
1.  compute a MST T of G ;
2.  select any vertex r be the root of
    the tree ;
3.  let L be the list of vertices
    visited in a preorder tree walk
    of T ;
4.  return the hamiltonian cycle H that
    visits the vertices in the order L ;
}
```

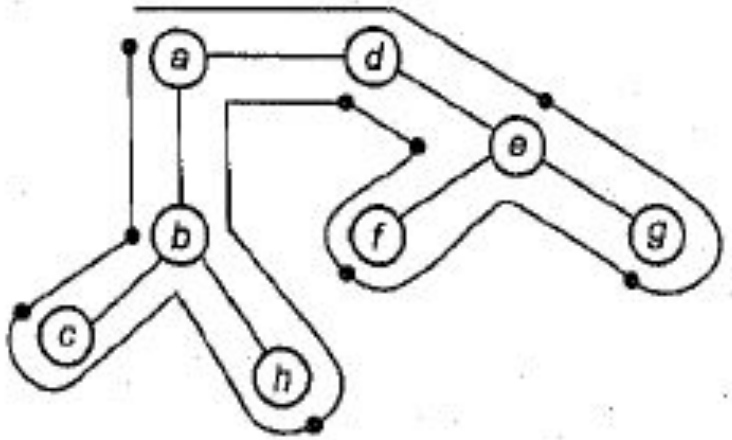




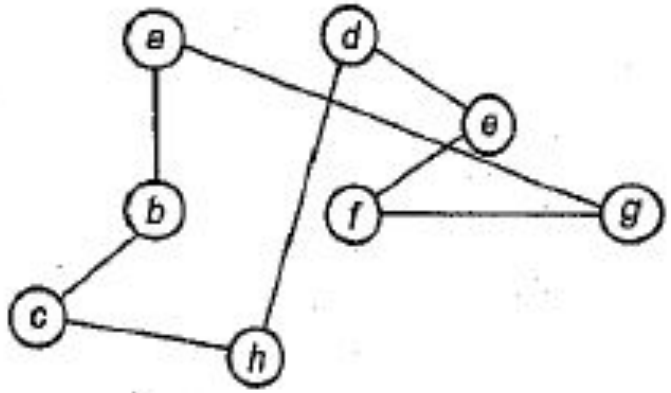
(1) A given set of points



(2) MST T



(3) Full tree walk on T.



(4) A preorder sequence gives a tour *H*.

Approx-TSP is a 2-approximation algorithm i.e.,

$$\frac{c(H)}{c(H^*)} \leq 2$$

Proof.

1. Observe that if we remove any edge from H^*, then it becomes to a spanning tree, hence we have $c(T) \leq c(H^*)$:
2. The cost of a full walk on T is $2c(T)$; since H makes a short-cut on the full walk, by *triangle inequality*, we have $c(H) \leq 2c(T)$;
3. Combining, we have

$$\frac{c(H)}{2} \leq c(T) \leq c(H^*) \quad \Rightarrow \quad \frac{c(H)}{c(H^*)} \leq 2$$

VERTEX COVER

VERTEX COVER APPR ALGO

Vertex Cover. A vertex cover of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices.

The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of vertex cover problem, *i.e.*, we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C^*.

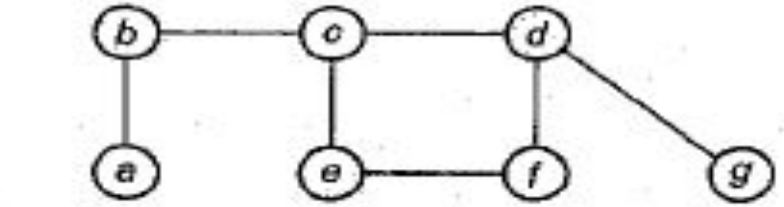


Figure 39.1

An approximate algorithm for Vertex Cover :

```
Approx-Vertex-Cover (G = (V, E) )
  {
      C = empty-set ;
      E' = E ;
      while E' is not empty do
      {
          let (u,v) be any edge in E' ; (*)
          add u and v to C ;
          remove from E' all edges incident to
          u or v ;
      }
      return C ;
}
```

VERTEX COVER 2 APPROXIMATION

Approx-Vertex-Cover is a 2-approximation algorithm, i.e.,

$$\frac{|C|}{|C^*|} \leq 2$$

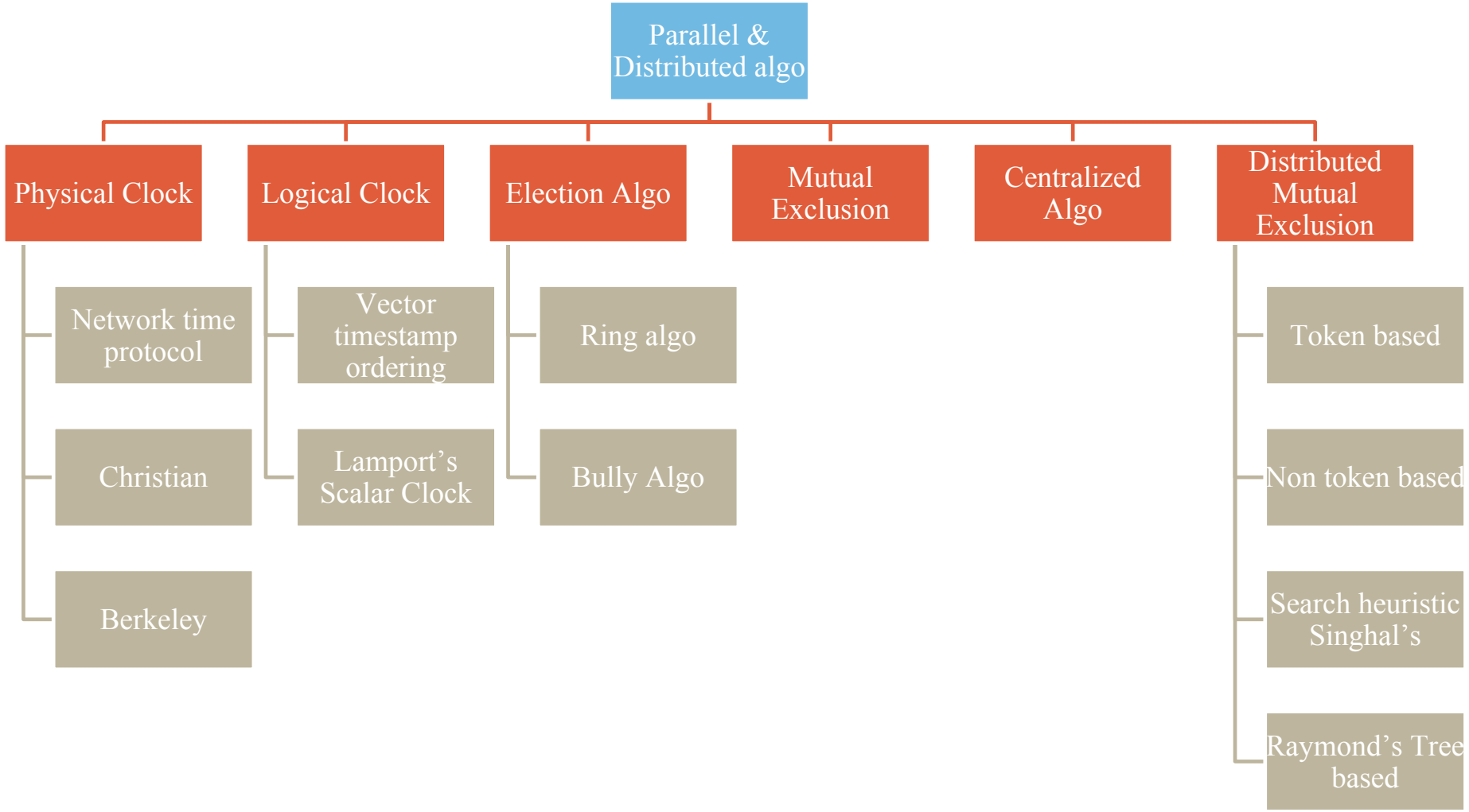
It means the number of vertices in C returned by Approx-Vertex-Cover guarantees to be at most twice of the optimal value.

Proof.

1. Let A be the edge set selected by line (*). Observe that $|C| = 2|A|$.
2. Observe that the edge in A does not have any common vertex between them. It means for $e = (x, y) \in A$, either x or y must be selected to the optimal vertex cover C^*. It follows $|C^*| \geq |A|$.
3. Now we have

$$\frac{|C|}{2} = |A| \leq |C^*| \quad \Rightarrow \quad \frac{|C|}{|C^*|} \leq 2.$$

PARALLEL & DISTRIBUTED ALGO



ELECTION ALGO

- Many distributed algorithms require a process to act as a coordinator.
- The coordinator can be any process that organizes actions of other processes.
- A coordinator may fail
- How is a new coordinator chosen or elected?

- Assumptions
  - Each process has a unique number to distinguish them.
  - Processes know each other's process number.

BULLY ALGO

- When a process, P, notices that the coordinator is no longer responding to requests, it initiates an election.
  - P sends an **ELECTION** message to all processes with higher numbers.
  - If no one responds, P wins the election and becomes a coordinator.
  - If one of the higher-ups answers, it takes over. P's job is done.

- When a process gets an ELECTION message from one of its lower-numbered colleagues:
  - Receiver sends an **OK** message back to the sender to indicate that he is alive and will take over.
  - Eventually, all processes give up apart of one, and that one is the new coordinator.
  - The new coordinator announces its victory by sending all processes a **CO-ORDINATOR** message telling them that it is the new coordinator.

- If a process that was previously down comes back:
  - It holds an election.
  - If it happens to be the highest process currently running, it will win the election and take over the coordinator's job.
- "Biggest guy" always wins and hence the name "bully" algorithm.

RING ALGO

Algorithm

- When a process notices that coordinator is not functioning:
  - Builds an **ELECTION** message (containing its own process number)
  - Sends the message to its successor (if successor is down, sender skips over it and goes to the next member along the ring, or the one after that, until a running process is located).
  - At each step, sender adds its own process number to the list in the message.

- Algorithm (continued)
  - When the message gets back to the process that started it all:

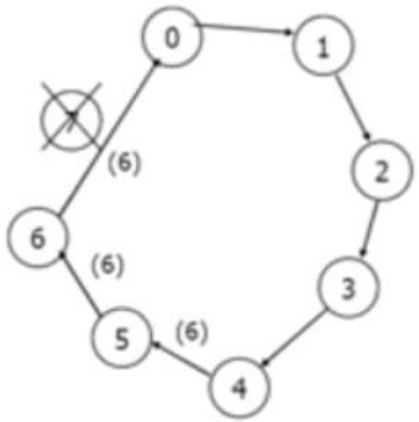
    . Message comes back to initiator, here the initiator is 4.

    In the queue the process with maximum Id Number wins

    Initiator announces the winner by sending another message around the ring

Initiation:

1. Process 4 sends an ELECTION message to its successor (or next alive process) with its ID



Leader Election:

3. Message comes back to initiator, here the initiator is 4.
4. Initiator announces the winner by sending another message around the ring

DISTRIBUTED MUTUAL EXCLUSIVE ALGO

- NON TOKEN BASED
  - Lamport’s Distributed Mutual Algo
  - Ricart AGRAWALA Algo
  - Maekawa’s Voting Algo
- TOKEN BASED
  - Suzuki Kasami broadcast algo
  - Singhal Heuristic Algo

CENTRALIZED ALGO