



BIJU PATNAIK UNIVERSITY OF TECHNOLOGY,  
ODISHA

Model  
Questions and Answers  
on

**PARALLEL COMPUTING**

Prepared by,  
Dr. Subhendu Kumar Rath,  
BPUT, Odisha.



## Model Questions and Answers

### Subject – Parallel Computing (Module-I,II,III)

By Dr. Subhendu Kumar Rath, BPUT

#### Short Questions:

1) **What is Parallel Computing?**

**Answer:** Parallel Computing resembles the study of designing algorithms such that the time complexity is minimum. Thus the speed up factor is taken into consideration.

2) **What is a parallel computer?**

**Answer:** A parallel Computer is simply a collection of processors, typically of the same type, interconnected in a certain fashion to allow the coordination of their activities and the exchange of data.

3) **What is Moore's Law?**

**Answer:** **Moore's Law** states that 'circuit complexity doubles every eighteen months'.

4) **What are the applications parallel computing?**

**Answer:** There are several applications of parallel computing.

- Effective parallel algorithms are required for problems such as association rule mining, clustering, classification, and time-series analysis.
- Bioinformatics and astrophysics
- Optimization and Branch-and-bound, and Genetic programming
- Association rule mining, clustering, classification, and time-series analysis.

5) **What is pipelining?**

**Answer:** **Pipelining** is a technique of dividing one task into multiple subtasks and executing the subtasks in parallel with multiple hardware units.

6) **What is super-scalar Execution?**

**Answer:** The ability of a processor to issue multiple instructions in the same cycle is referred to as **superscalar execution**.

7) **What is true data dependency?**

**Answer:** If the results of an instruction is required for subsequent instructions then this is referred to as **true data dependency**.

8) **What is resource dependency?**

**Answer:** The dependency in which two instructions compete for a single processor resource is referred to as **resource dependency**.

9) **What is Horizontal Waste?**

**Answer:** If only part of the execution units is used during a cycle, it is termed **horizontal waste**.

10) **What is Vertical Waste?**

**Answer:** If, during a particular cycle, no instructions are issued on the execution units, it is referred to as **vertical waste**.

11) **What is VLIW?**

**Answer:** Instructions that can be executed concurrently are packed into groups and parceled off to the processor as a single long instruction word to be executed on multiple functional units at



the same time. This is called very large instruction word (VLIW).

**12) What is bandwidth of memory?**

**Answer:** The rate at which data can be pumped from the memory to the processor determines the **bandwidth** of the memory system.

**13) What is latency of memory?**

**Answer:** The time taken in putting a request for a memory word and returns a block of data of size  $b$  containing the requested word is referred to as the **latency** of the memory.

**14) What is logical and physical organization of parallel programming platforms?**

**Answer:** The logical organization refers to a programmer's view of the platform while the physical organization refers to the actual hardware organization of the platform.

**15) Differentiate between SIMD and MIMD?**

**Answer:** In architectures referred to as single instruction stream, multiple data stream (SIMD), a single control unit dispatches instructions to each processing unit. In an SIMD parallel computer, the same instruction is executed synchronously by all processing units.

Computers in which each processing element is capable of executing a different program independent of the other processing elements are called multiple instruction stream, multiple data stream (MIMD) computers.

SIMD computers require less hardware than MIMD computers because they have only one global control unit.

SIMD computers require less memory because only one copy of the program needs to be stored.

In contrast, MIMD computers store the program and operating system at each processor.

**16) Differentiate between UMA and NUMA.**

**Answer:** If the time taken by a processor to access any memory word in the system (global or local) is identical, the platform is classified as a uniform memory access (UMA) multicomputer.

On the other hand, if the time taken to access certain memory words is longer than others, the platform is called a non-uniform memory access (NUMA) multicomputer.

**17) What is Cache Coherence?**

**Answer:** In shared address space platform ensuring that concurrent operations on multiple copies of the same memory word have well-defined semantics is called cache coherence.

**18) What is PRAM Model?**

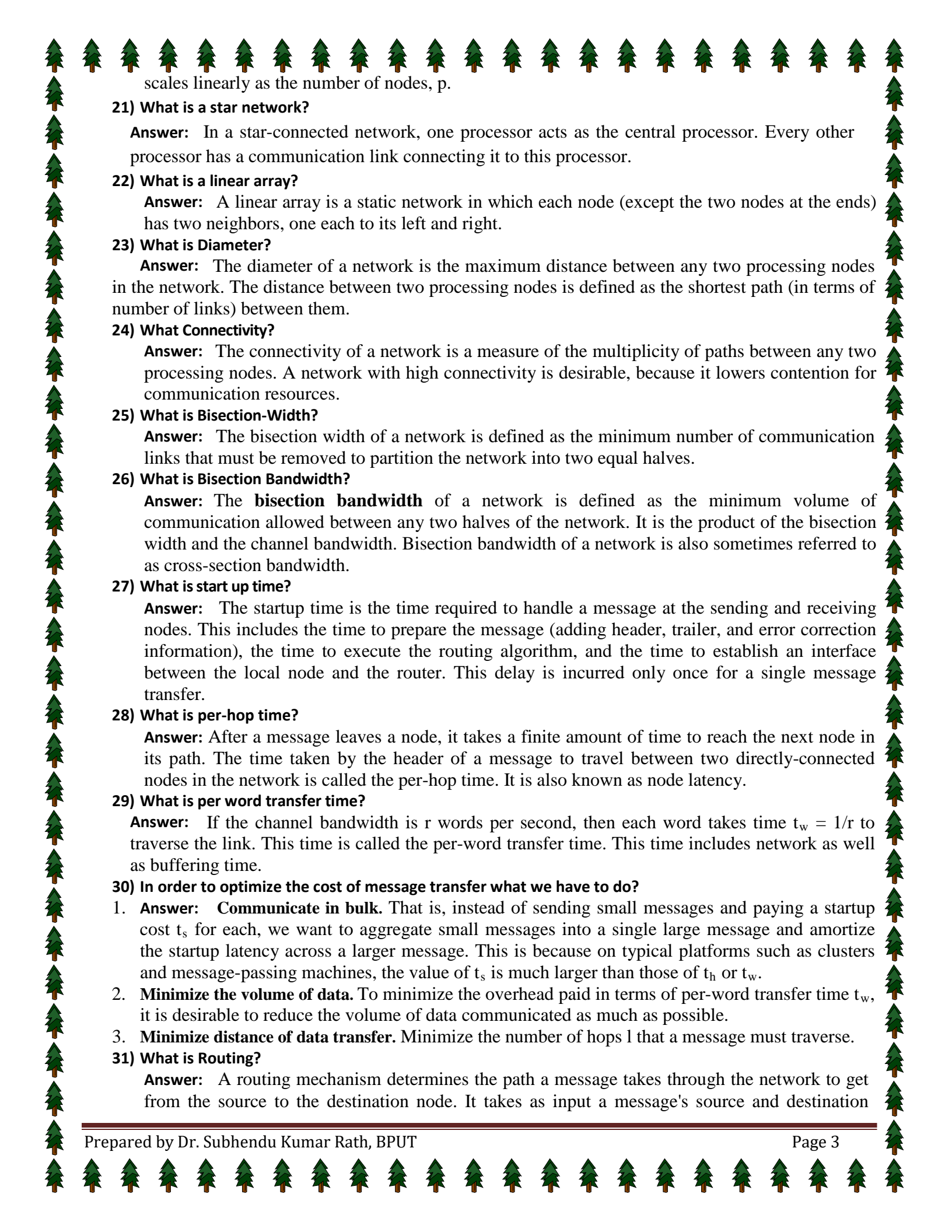
**Answer:** A Model of computation (the Random Access Machine, or RAM) consists of  $p$  processors and a global memory of unbounded size that is uniformly accessible to all processors. All processors access the same address space. Processors share a common clock but may execute different instructions in each cycle. This ideal model is also referred to as a parallel random access machine (PRAM).

**19) Differentiate between Static and Dynamic Network.**

**Answer:** Static networks consist of point-to-point communication links among processing nodes and are also referred to as direct networks. Dynamic networks, on the other hand, are built using switches and communication links.

**20) What is Bus-based Network?**

**Answer:** A bus-based network is perhaps the simplest network consisting of a shared medium that is common to all the nodes. A bus has the desirable property that the cost of the network



scales linearly as the number of nodes,  $p$ .

**21) What is a star network?**

**Answer:** In a star-connected network, one processor acts as the central processor. Every other processor has a communication link connecting it to this processor.

**22) What is a linear array?**

**Answer:** A linear array is a static network in which each node (except the two nodes at the ends) has two neighbors, one each to its left and right.

**23) What is Diameter?**

**Answer:** The diameter of a network is the maximum distance between any two processing nodes in the network. The distance between two processing nodes is defined as the shortest path (in terms of number of links) between them.

**24) What Connectivity?**

**Answer:** The connectivity of a network is a measure of the multiplicity of paths between any two processing nodes. A network with high connectivity is desirable, because it lowers contention for communication resources.

**25) What is Bisection-Width?**

**Answer:** The bisection width of a network is defined as the minimum number of communication links that must be removed to partition the network into two equal halves.

**26) What is Bisection Bandwidth?**

**Answer:** The **bisection bandwidth** of a network is defined as the minimum volume of communication allowed between any two halves of the network. It is the product of the bisection width and the channel bandwidth. Bisection bandwidth of a network is also sometimes referred to as cross-section bandwidth.

**27) What is start up time?**

**Answer:** The startup time is the time required to handle a message at the sending and receiving nodes. This includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local node and the router. This delay is incurred only once for a single message transfer.

**28) What is per-hop time?**

**Answer:** After a message leaves a node, it takes a finite amount of time to reach the next node in its path. The time taken by the header of a message to travel between two directly-connected nodes in the network is called the per-hop time. It is also known as node latency.

**29) What is per word transfer time?**

**Answer:** If the channel bandwidth is  $r$  words per second, then each word takes time  $t_w = 1/r$  to traverse the link. This time is called the per-word transfer time. This time includes network as well as buffering time.

**30) In order to optimize the cost of message transfer what we have to do?**

- 1. Answer: Communicate in bulk.** That is, instead of sending small messages and paying a startup cost  $t_s$  for each, we want to aggregate small messages into a single large message and amortize the startup latency across a larger message. This is because on typical platforms such as clusters and message-passing machines, the value of  $t_s$  is much larger than those of  $t_h$  or  $t_w$ .
- 2. Minimize the volume of data.** To minimize the overhead paid in terms of per-word transfer time  $t_w$ , it is desirable to reduce the volume of data communicated as much as possible.
- 3. Minimize distance of data transfer.** Minimize the number of hops  $l$  that a message must traverse.

**31) What is Routing?**

**Answer:** A routing mechanism determines the path a message takes through the network to get from the source to the destination node. It takes as input a message's source and destination

nodes. It may also use information about the state of the network. It returns one or more paths through the network from the source to the destination node.

**32) What is store-and forward routing?**

**Answer:** In store-and-forward routing, when a message is traversing a path with multiple links, each intermediate node on the path forwards the message to the next node after it has received and stored the entire message.

**33) What is Snoopy cache system?**

**Answer:** Snoopy caches are typically associated with multiprocessor systems based on broadcast interconnection networks such as a bus or a ring. In such systems, all processors snoop on (monitor) the bus for transactions. This allows the processor to make state transitions for its cache-blocks.

**34) What is False Sharing?**

**Answer:** False sharing refers to the situation in which different processors update different parts of the same cache-line.

**35) What are the protocols used in the cache coherence system?**

**Answer:** The protocols used in cache coherence system are (a) Invalidate protocol and (b) Update protocol.

**36) What is Routing mechanism?**

**Answer:** A routing mechanism determines the path a message takes through the network to get from the source to the destination node.

**37) What are the classifications of Routing mechanism?**

**Answer:** Routing mechanisms can be classified as **minimal or non-minimal**. A **minimal routing** mechanism always selects one of the shortest paths between the source and the destination. In a minimal routing scheme, each link brings a message closer to its destination, but the scheme can lead to congestion in parts of the network. A **non-minimal routing** scheme, in contrast, may route the message along a longer path to avoid network congestion.

**38) Differentiate between deterministic routing and adaptive routing.**

**Answer:** A **deterministic routing** scheme determines a unique path for a message, based on its source and destination. It does not use any information regarding the state of the network. Deterministic schemes may result in uneven use of the communication resources in a network. In contrast, an **adaptive routing** scheme uses information regarding the current state of the network to determine the path of the message. Adaptive routing detects congestion in the network and routes messages around it.

**39) Define congestion, dilation and expansion of mapping of a graph.**

**Answer:** The maximum number of edges mapped onto any edge in  $E'$  is called the **congestion** of the mapping.

The maximum number of links in  $E'$  that any edge in  $E$  is mapped onto is called the **dilation** of the mapping.

The ratio of the number of nodes in the set  $V'$  to that in set  $V$  is called the **expansion** of the mapping.

**40) Define Decomposition and Tasks**

**Answer:** The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel, is called **decomposition**. **Tasks** are programmer-defined units of computation into which the main computation is subdivided by means of decomposition.

**41) What is a Task-dependency graph?**

**Answer:** A **task-dependency graph** is a directed acyclic graph in which the nodes represent tasks and the directed edges indicate the dependencies amongst them. The task corresponding to a node can be

executed when all tasks connected to this node by incoming edges have completed. Note that task-dependency graphs can be disconnected and the edge-set of a task-dependency graph can be empty.

**42) Define granularity, fine-grained and coarse-grained of a decomposition.**

**Answer:** The number and size of tasks into which a problem is decomposed determines the **granularity** of the decomposition. Decomposition into a large number of small tasks is called **fine-grained** and decomposition into a small number of large tasks is called **coarse-grained**.

**43) Define maximum degree of concurrency and average degree of concurrency.**

**Answer:** The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its **maximum degree of concurrency**.

The **average degree of concurrency** is the average number of tasks that can run concurrently over the entire duration of execution of the program.

**44) What is critical path and critical path length?**

**Answer:** The longest directed path between any pair of start and finish nodes is known as the **critical path**. The sum of the weights of nodes along this path is known as the **critical path length**, where the weight of a node is the size or the amount of work associated with the corresponding task.

**45) Differentiate between processes and processors.**

**Answer:** **Processes** are logical computing agents that perform tasks. **Processors** are the hardware units that physically perform computations.

**46) What is Owner-Computes rule?**

**Answer:** A decomposition based on partitioning output or input data is also widely referred to as the **Owner-computes rule**.

**47) Differentiate between One-to-All Broadcast and All-to-One Reduction.**

**Answer:** Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them. This operation is known as **one-to-all broadcast**. Initially, only the source process has the data of size  $m$  that needs to be broadcast. At the termination of the procedure, there are  $p$  copies of the initial data – one belonging to each process. The dual of one-to-all broadcast is **all-to-one reduction**. In an all-to-one reduction operation, each of the  $p$  participating processes starts with a buffer  $M$  containing  $m$  words. The data from all processes are combined through an associative operator and accumulated at a single destination process into one buffer of size  $m$ .

**48) What is a Parallel system? What are the sources of overhead in parallel programs?**

**Answer:** A **parallel system** is the combination of an algorithm and the parallel architecture on which it is implemented.

**Interprocess Interaction:** Any nontrivial parallel system requires its processing elements to interact and communicate data (e.g., intermediate results). The time spent communicating data between processing elements is usually the most significant source of parallel processing overhead.

**Idling:** Processing elements in a parallel system may become idle due to many reasons such as load imbalance, synchronization, and presence of serial components in a program.

**Excess Computation:** The difference in computation performed by the parallel program and the best serial program is the excess computation overhead incurred by the parallel program.

**49) What are the performance metrics of parallel systems?**

**Answer:** **Execution Time:** The serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The parallel runtime is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes execution. We denote the serial runtime by  $T_s$  and the parallel runtime by  $T_p$ .

**Total Parallel Overhead**

We define overhead function or total overhead of a parallel system as the total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element. We denote the overhead function of a parallel system by the symbol  $T_o$ .



### Speedup

It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with  $p$  identical processing elements. We denote speedup by the symbol  $S$ .

### Efficiency

Efficiency is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to  $p$  and efficiency is equal to one. We denote efficiency by the symbol  $E$ . Mathematically, it is given by

$$E = \frac{S}{p}$$

### 50) What is isoefficiency function?

**Answer:** From the equation  $W = KT_o(W, p)$ , the problem size  $W$  can usually be obtained as a function of  $p$  by algebraic manipulations. This function dictates the growth rate of  $W$  required to keep the efficiency fixed as  $p$  increases. We call this function the **isoefficiency function** of the parallel system. The isoefficiency function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processing elements.

### 51) What is Binary reflected Gray code?

**Answer:** The function  $G(i, x)$  is defined as follows:

$$G(0, 1) = 0$$
$$G(1, 1) = 1$$
$$G(i, x + 1) = \begin{cases} G(i, x), & i < 2^x \\ 2^x + G(2^{x+1} - 1 - i, x), & i \geq 2^x \end{cases}$$

The function  $G$  is called the binary reflected Gray code (RGC).

### 52) State Amdahl's law.

**Answer:** Amdahl's law states that the overall speedup of applying the improvement will be:

$$\frac{1}{(1 - P) + \frac{P}{S}}$$

To see how this formula was derived, assume that the running time of the old computation was 1, for some unit of time. The running time of the new computation will be the length of time the unimproved fraction takes (which is  $1 - P$ ), plus the length of time the improved fraction takes. The length of time for the improved part of the computation is the length of the improved part's former running time divided by the speedup, making the length of time of the improved part  $P/S$ . The final speedup is computed by dividing the old running time by the new running time, which is what the above formula does.

### 53) What is MPI?

**Answer:** **Message Passing Interface (MPI)** is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in the C programming language.

### 54) What is scalability of a parallel system?

**Answer:** The scalability of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processing elements.

55) What is data parallel algorithm model?

**Answer:** The data-parallel model is one of the simplest algorithm models. In this model, the tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data. This type of parallelism that is a result of identical operations being applied concurrently on different data items is called data parallelism.

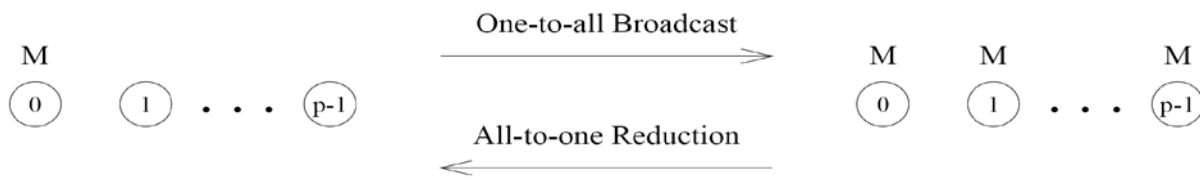
56) What do you mean by speculative decomposition of parallel program?

**Answer:** Speculative decomposition is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. In this situation, while one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage.

57) Differentiate between one-to-all broadcast and all-to-one reduction.

**Answer:** Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them. This operation is known as **one-to-all broadcast**. Initially, only the source process has the data of size  $m$  that needs to be broadcast. At the termination of the procedure, there are  $p$  copies of the initial data – one belonging to each process. The dual of one-to-all broadcast is **all-to-one reduction**. In an all-to-one reduction operation, each of the  $p$  participating processes starts with a buffer  $M$  containing  $m$  words. The data from all processes are combined through an associative operator and accumulated at a single destination process into one buffer of size  $m$ . Reduction can be used to find the sum, product, maximum, or minimum of sets of numbers – the  $i$ th word of the accumulated  $M$  is the sum, product, maximum, or minimum of the  $i$ th words of each of the original buffers. The figure shows one-to-all broadcast and all-to-one reduction among  $p$  processes.

Figure 1. One-to-all broadcast and all-to-one reduction.



One-to-all broadcast and all-to-one reduction are used in several important parallel algorithms including matrix-vector multiplication, Gaussian elimination, shortest paths, and vector inner product. In the following subsections, we consider the implementation of one-to-all broadcast in detail on a variety of interconnection topologies.





## Long Questions:

### 58) What is pipelining? Describe the speed up gain due to pipelining.

**Answer: Pipelining** is a technique of dividing one task into multiple subtasks and executing the subtasks in parallel with multiple hardware units.

In a pipelining processor, multiple instructions are executed at various stages of instruction cycle simultaneously in different sections of the processor.

Ex: In pipelining processors,  $S_1, S_2, S_3, \dots, S_n$  are  $n$  sections that form the pipeline. Each section performs a subtask on the input received from the interstage buffer that stores the output of the previous section. All sections can perform their operations simultaneously.

The objective of pipelining processing is to increase throughput due to the overlap between the execution of the consecutive task.

An instruction pipeline divides an instruction cycle actions into multiple steps that are executed one-by-one in different sections of the processor.

The number of sections in the pipeline is designed by the computer architect.

Consider the instruction cycle is divided into four steps:

a) Instruction Fetch (IF)

b) Instruction Decode (ID)

c) Execute (EX)

d) Write Result (WR)

Assume that there are  $m$  instructions and these instructions will be executed in  $n$ -sections of the pipeline processor.

The time taken for the first instruction =  $nt_c$ , where  $t_c$  is the duration of the clock cycle.

Time taken for remaining  $(m-1)$  instructions =  $(m-1)t_c$

Total time taken for  $m$  instructions =  $nt_c + (m-1)t_c = (n+m-1)t_c$

If the processor is non-pipelined, then the total time taken for  $m$  instructions is  $mnt_c$ .

Hence performance gain due to pipeline =  $mnt_c / (m+n-1)t_c$

### 59) Explain Super-Scalar execution with the help of an example.

**Answer:** Consider a processor with two pipelines and the ability to simultaneously issue two instructions. These processors are sometimes also referred to as **super-pipelined processors**. The ability of a processor to issue multiple instructions in the same cycle is referred to as **superscalar execution**. Since the architecture illustrated in the following figure allows two issues per clock cycle, it is also referred to as two-way superscalar or dual issue execution.

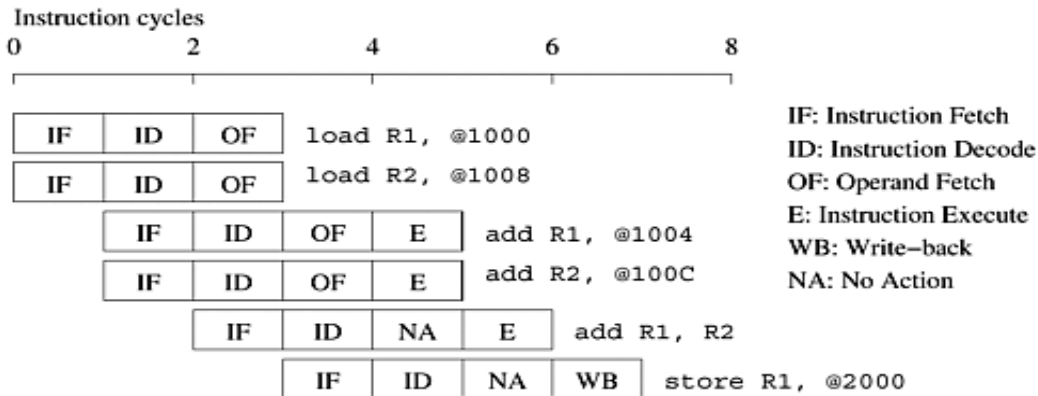
Consider the execution of the first code fragment in the figure for adding four numbers. The first and second instructions are independent and therefore can be issued concurrently. This is illustrated in the simultaneous issue of the instructions `load R1, @1000` and `load R2, @1008` at  $t = 0$ . The instructions are fetched, decoded, and the operands are fetched. The next two instructions, `add R1, @1004` and `add R2, @100C` are also mutually independent, although they must be executed after the first two instructions. Consequently, they can be issued concurrently at  $t = 1$  since the processors are pipelined. These instructions terminate at  $t = 5$ . The next two instructions, `add R1, R2` and `store R1, @2000` cannot be executed concurrently since the result of the former (contents of register R1) is used by the latter. Therefore, only the `add` instruction is issued at  $t = 2$  and the `store` instruction at  $t = 3$ . Note that the instruction `add R1, R2` can be executed only after the previous two instructions have been executed.

Figure : Example of a two-way superscalar execution of instructions.

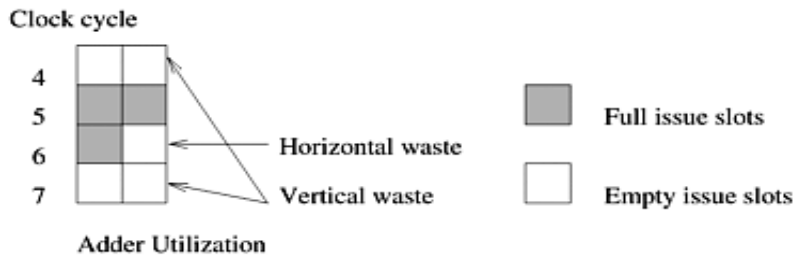
- |                    |                    |                    |
|--------------------|--------------------|--------------------|
| 1. load R1, @1000  | 1. load R1, @1000  | 1. load R1, @1000  |
| 2. load R2, @1008  | 2. add R1, @1004   | 2. add R1, @1004   |
| 3. add R1, @1004   | 3. add R1, @1008   | 3. load R2, @1008  |
| 4. add R2, @100C   | 4. add R1, @100C   | 4. add R2, @100C   |
| 5. add R1, R2      | 5. store R1, @2000 | 5. add R1, R2      |
| 6. store R1, @2000 |                    | 6. store R1, @2000 |

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

### 60) What is PRAM Model? What are the subclasses of PRAM?

**Answer:** A natural extension of the serial model of computation (the Random Access Machine, or RAM) consists of  $p$  processors and a global memory of unbounded size that is uniformly accessible to all processors. All processors access the same address space. Processors share a common clock but may execute different instructions in each cycle. This ideal model is also referred to as a parallel random access machine (PRAM). Since PRAMs allow concurrent access to various memory locations, depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.

1. **Exclusive-read, exclusive-write (EREW) PRAM.** In this class, access to a memory location is exclusive. No concurrent read or write operations are allowed. This is the weakest PRAM model, affording minimum concurrency in memory access.
2. **Concurrent-read, exclusive-write (CREW) PRAM.** In this class, multiple read accesses to a memory location are allowed. However, multiple write accesses to a memory location are serialized.
3. **Exclusive-read, concurrent-write (ERCW) PRAM.** Multiple write accesses are allowed to a memory location, but multiple read accesses are serialized.

4. **Concurrent-read, concurrent-write (CRCW) PRAM.** This class allows multiple read and writes accesses to a common memory location. This is the most powerful PRAM model.

Allowing concurrent read access does not create any semantic discrepancies in the program. However, concurrent write access to a memory location requires arbitration. Several protocols are used to resolve concurrent writes. The most frequently used protocols are as follows:

- **Common**, in which the concurrent write is allowed if all the values that the processors are attempting to write are identical.
- **Arbitrary**, in which an arbitrary processor is allowed to proceed with the write operation and the rest fail.
- **Priority**, in which all processors are organized into a predefined prioritized list, and the processor with the highest priority succeeds and the rest fail.
- **Sum**, in which the sum of all the quantities is written (the sum-based write conflict resolution model can be extended to any associative operator defined on the quantities being written).

61) Explain Bus-based, multi-stage and crossbar network topologies.

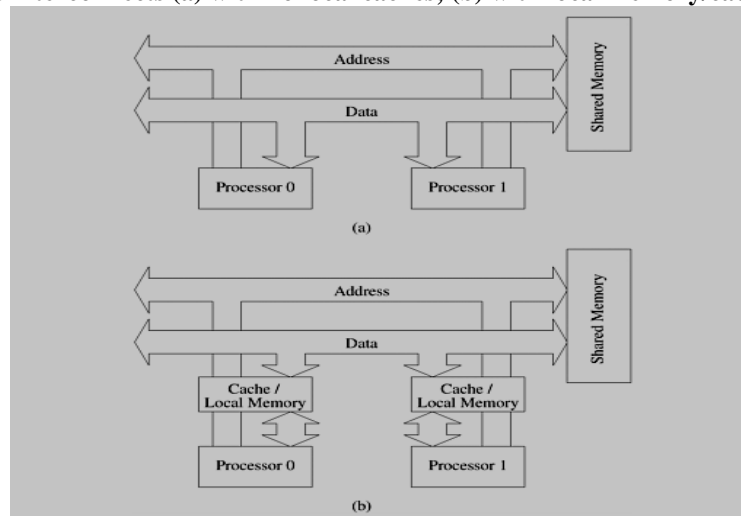
**Answer:** A wide variety of network topologies have been used in interconnection networks. These topologies try to trade off cost and scalability with performance. While pure topologies have attractive mathematical properties, in practice interconnection networks tend to be combinations or modifications of the pure topologies discussed in this section.

### Bus-Based Networks

A bus-based network is perhaps the simplest network consisting of a shared medium that is common to all the nodes. A bus has the desirable property that the cost of the network scales linearly as the number of nodes,  $p$ . This cost is typically associated with bus interfaces. Furthermore, the distance between any two nodes in the network is constant ( $O(1)$ ). Buses are also ideal for broadcasting information among nodes. Since the transmission medium is shared, there is little overhead associated with broadcast compared to point-to-point message transfer. However, the bounded bandwidth of a bus places limitations on the overall performance of the network as the number of nodes increases. Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessors are examples of such architectures.

It is possible to provide a cache for each node. Private data is cached at the node and only remote data is accessed through the bus.

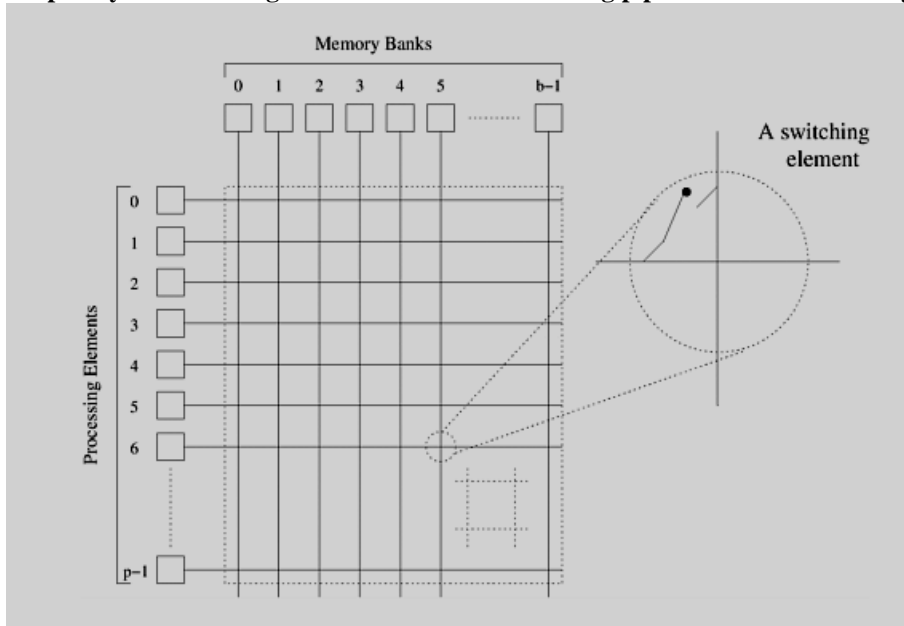
Figure Bus-based interconnects (a) with no local caches; (b) with local memory/caches.



### Crossbar Networks

A simple way to connect  $p$  processors to  $b$  memory banks is to use a crossbar network. A crossbar network employs a grid of switches or switching nodes as shown in the figure. The crossbar network is a non-blocking network in the sense that the connection of a processing node to a memory bank does not block the connection of any other processing nodes to other memory banks.

Figure . A completely non-blocking crossbar network connecting  $p$  banks to  $b$  memory banks.



The total number of switching nodes required to implement such a network is  $\Theta(pb)$ . It is reasonable to assume that the number of memory banks  $b$  is at least  $p$ ; otherwise, at any given time, there will be some processing nodes that will be unable to access any memory banks. Therefore, as the value of  $p$  is increased, the complexity (component count) of the switching network grows as  $\Omega(p^2)$ . (See the Appendix for an explanation of the  $\Omega$  notation.) As the number of processing nodes becomes large, this switch complexity is difficult to realize at high data rates. Consequently, crossbar networks are not very scalable in terms of cost.

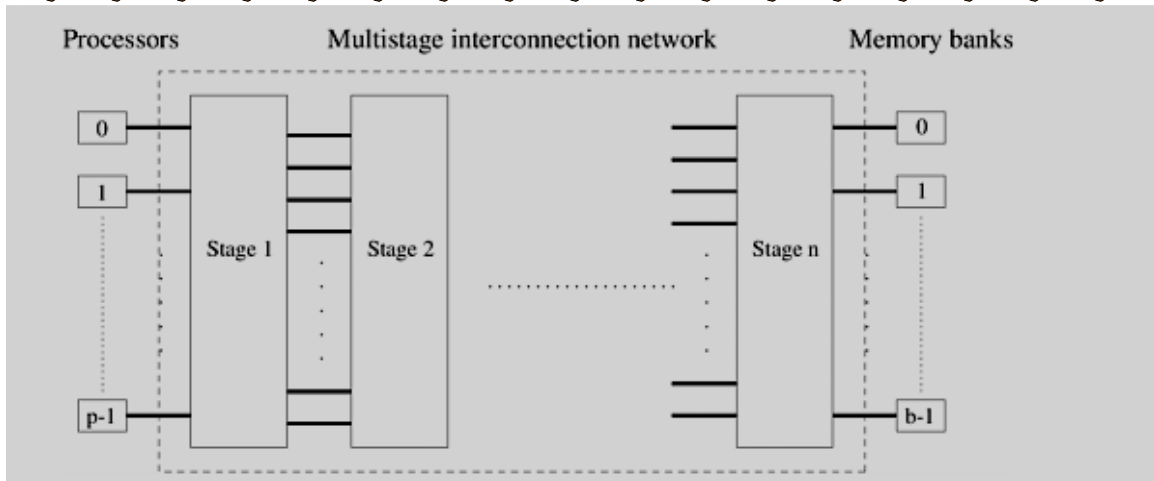
### Multistage Networks

The crossbar interconnection network is scalable in terms of performance but unscalable in terms of cost. Conversely, the shared bus network is scalable in terms of cost but unscalable in terms of performance. An intermediate class of networks called multistage interconnection networks lies between these two extremes. It is more scalable than the bus in terms of performance and more scalable than the crossbar in terms of cost.

The general schematic of a multistage network consisting of  $p$  processing nodes and  $b$  memory banks is shown in the figure. A commonly used multistage connection network is the omega network. This network consists of  $\log p$  stages, where  $p$  is the number of inputs (processing nodes) and also the number of outputs (memory banks). Each stage of the omega network consists of an interconnection pattern that connects  $p$  inputs and  $p$  outputs; a link exists between input  $i$  and output  $j$  if the following is true:

$$\text{Equation 1 : } \begin{aligned} j &= 2i, \text{ if } 0 \leq i \leq p/2 - 1 \\ j &= 2i + 1 - p, \text{ if } p/2 \leq i \leq p - 1 \end{aligned}$$

Figure . The schematic of a typical multistage interconnection network.



Equation 1 represents a left-rotation operation on the binary representation of  $i$  to obtain  $j$ . This interconnection pattern is called a perfect shuffle. Figure shows a perfect shuffle interconnection pattern for eight inputs and outputs. At each stage of an omega network, a perfect shuffle interconnection pattern feeds into a set of  $p/2$  switches or switching nodes. Each switch is in one of two connection modes. In one mode, the inputs are sent straight through to the outputs, as shown in Figure (a). This is called the pass-through connection. In the other mode, the inputs to the switching node are crossed over and then sent out, as shown in the figure (b). This is called the cross-over connection.

Figure . A perfect shuffle interconnection for eight inputs and outputs.

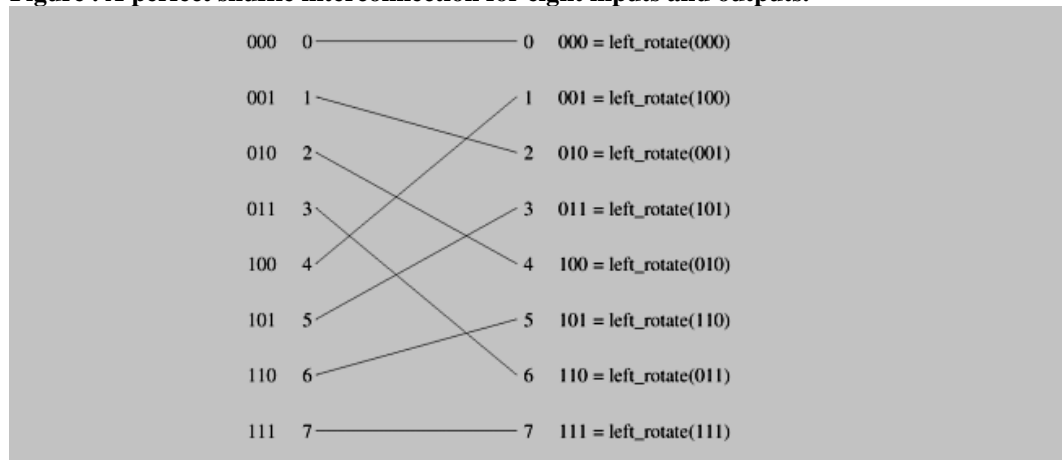
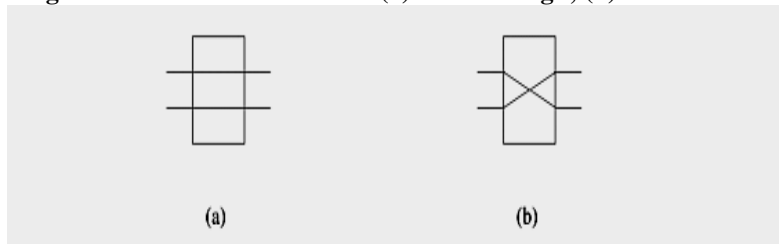


Figure . Two switching configurations of the 2 x 2 switch: (a) Pass-through; (b) Cross-over.



An omega network has  $p/2 \times \log p$  switching nodes, and the cost of such a network grows as  $\Theta(p \log p)$ . Note that this cost is less than the  $\Theta(p^2)$  cost of a complete crossbar network. Figure shows an omega network for eight processors (denoted by the binary numbers on the left) and eight memory banks (denoted by the binary numbers on the right). Routing data in an omega network is accomplished using a simple scheme. Let  $s$  be the binary representation of a processor that needs to write some data into memory bank  $t$ . The data traverses the link to the first switching node. If the most significant bits of  $s$  and

t are the same, then the data is routed in pass-through mode by the switch. If these bits are different, then the data is routed through in crossover mode. This scheme is repeated at the next switching stage using the next most significant bit. Traversing  $\log p$  stages uses all  $\log p$  bits in the binary representations of  $s$  and  $t$ .

Figure . A complete omega network connecting eight inputs and eight outputs.

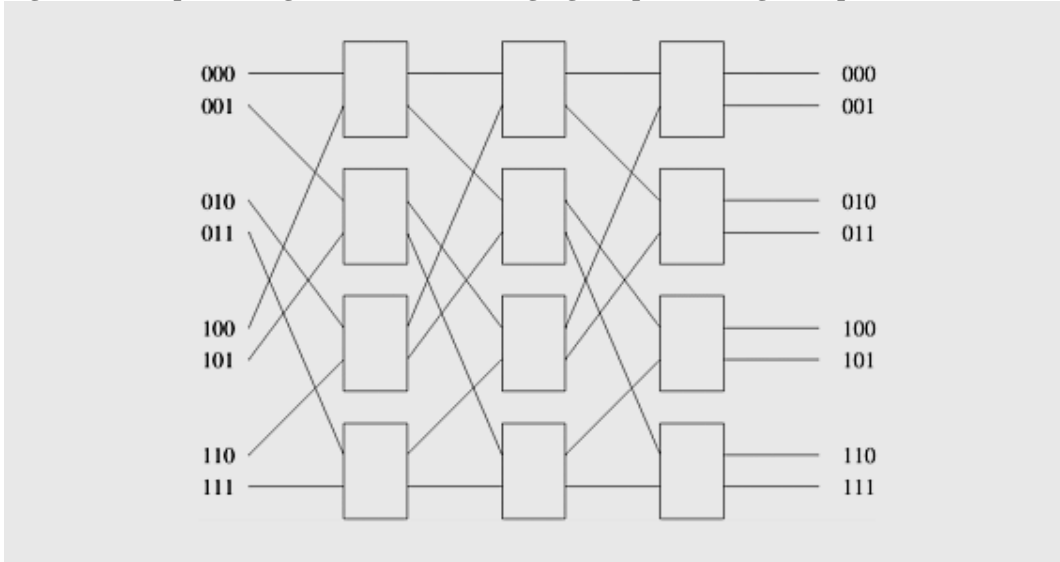


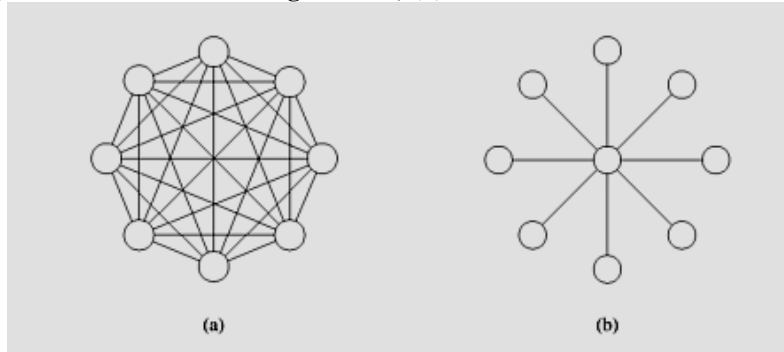
Figure shows data routing over an omega network from processor two (010) to memory bank seven (111) and from processor six (110) to memory bank four (100). This figure also illustrates an important property of this network. When processor two (010) is communicating with memory bank seven (111), it blocks the path from processor six (110) to memory bank four (100). Communication link AB is used by both communication paths. Thus, in an omega network, access to a memory bank by a processor may disallow access to another memory bank by another processor. Networks with this property are referred to as blocking networks.

**62) Explain completely-connected, star, linear array and Mesh networks?**

**Answer: Completely-Connected Network**

In a completely-connected network, each node has a direct communication link to every other node in the network. Figure (a) illustrates a completely-connected network of eight nodes. This network is ideal in the sense that a node can send a message to another node in a single step, since a communication link exists between them. Completely-connected networks are the static counterparts of crossbar switching networks, since in both networks; the communication between any input/output pair does not block communication between any other pair.

Figure : (a) A completely-connected network of eight nodes; (b) a star connected network of nine nodes.



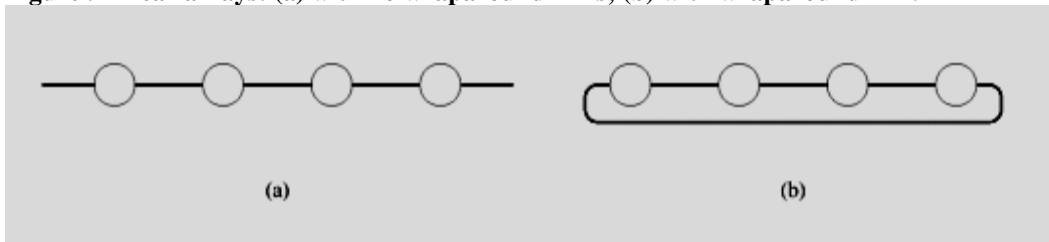
**Star-Connected Network**

In a star-connected network, one processor acts as the central processor. Every other processor has a communication link connecting it to this processor. Figure (b) shows a star-connected network of nine processors. The star-connected network is similar to bus-based networks. Communication between any pair of processors is routed through the central processor, just as the shared bus forms the medium for all communication in a bus-based network. The central processor is the bottleneck in the star topology.

**Linear Arrays, Meshes, and k-d Meshes**

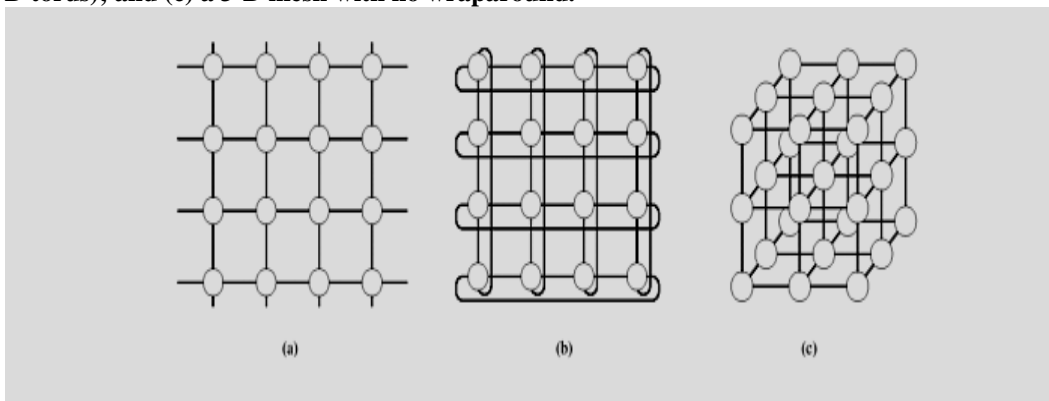
Due to the large number of links in completely connected networks, sparser networks are typically used to build parallel computers. A family of such networks spans the space of linear arrays and hypercubes. A linear array is a static network in which each node (except the two nodes at the ends) has two neighbors, one each to its left and right. A simple extension of the linear array (Figure (a)) is the ring or a 1-D torus (Figure (b)). The ring has a wraparound connection between the extremities of the linear array. In this case, each node has two neighbors.

Figure . Linear arrays: (a) with no wraparound links; (b) with wraparound link.



A two-dimensional mesh illustrated in the figure is an extension of the linear array to two-dimensions. Each dimension has  $\sqrt{p}$  nodes with a node identified by a two-tuple (i, j). Every node (except those on the periphery) is connected to four other nodes whose indices differ in any dimension by one. A 2-D mesh has the property that it can be laid out in 2-D space, making it attractive from a wiring standpoint. Furthermore, a variety of regularly structured computations map very naturally to a 2-D mesh. For this reason, 2-D meshes were often used as interconnects in parallel machines. Two dimensional meshes can be augmented with wraparound links to form two dimensional tori illustrated in the figure. The three-dimensional cube is a generalization of the 2-D mesh to three dimensions, as illustrated in Figure (c). Each node element in a 3-D cube, with the exception of those on the periphery, is connected to six other nodes, two along each of the three dimensions. A variety of physical simulations commonly executed on parallel computers (for example, 3-D weather modeling, structural modeling, etc.) can be mapped naturally to 3-D network topologies. For this reason, 3-D cubes are used commonly in interconnection networks for parallel computers (for example, in the Cray T3E).

Figure : Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

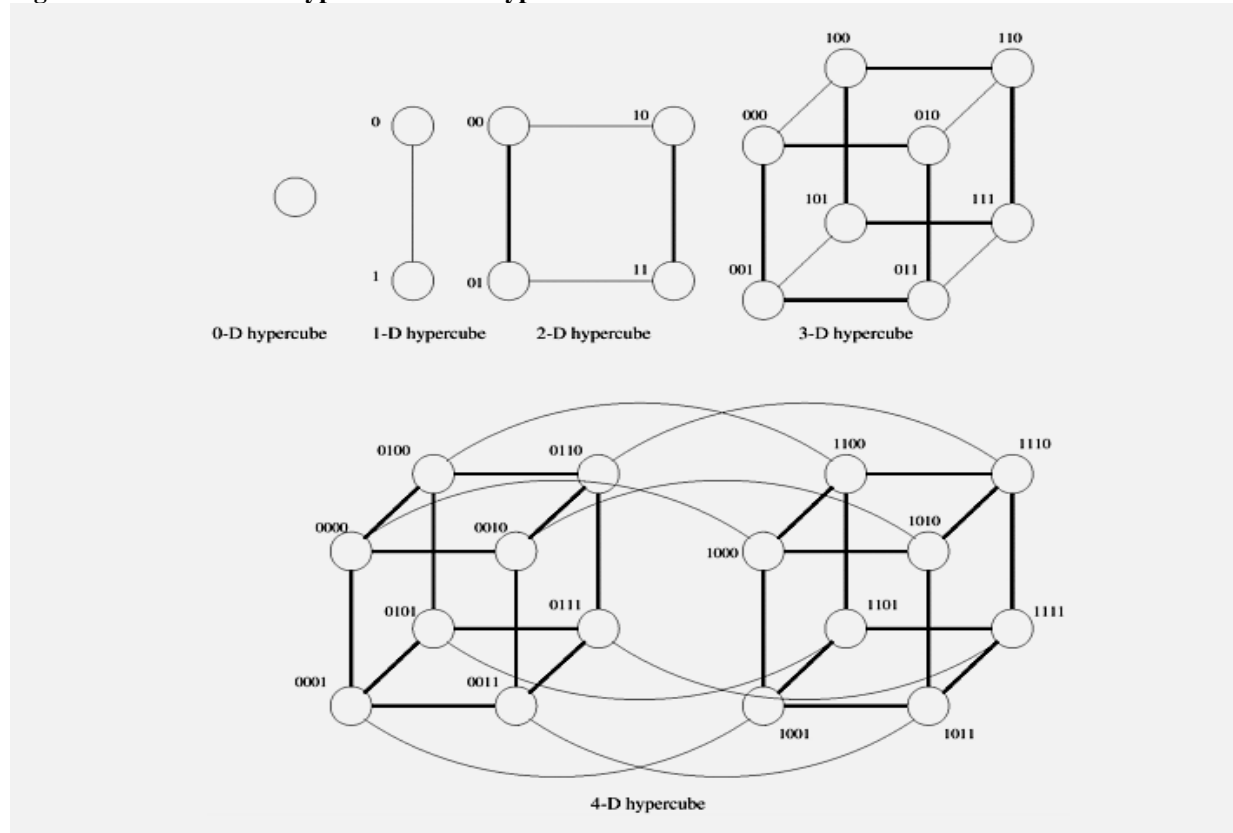


The general class of k-d meshes refers to the class of topologies consisting of d dimensions with k nodes along each dimension.

**63) Explain the construction of hypercube from lower dimensions and tree based network.**

**Answer:** Just as a linear array forms one extreme of the k-d mesh family, the other extreme is formed by an interesting topology called the hypercube. The hypercube topology has two nodes along each dimension and  $\log p$  dimensions. The construction of a hypercube is illustrated in the figure. A zero-dimensional hypercube consists of  $2^0$ , i.e., one node. A one-dimensional hypercube is constructed from two zero-dimensional hypercubes by connecting them. A two-dimensional hypercube of four nodes is constructed from two one-dimensional hypercubes by connecting corresponding nodes. In general a d-dimensional hypercube is constructed by connecting corresponding nodes of two (d - 1) dimensional hypercubes. The following figure illustrates this for up to 16 nodes in a 4-D hypercube.

**Figure . Construction of hypercubes from hypercubes of lower dimension.**



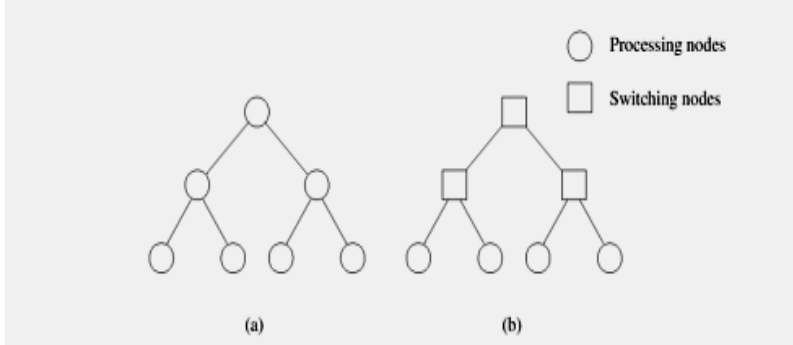
It is useful to derive a numbering scheme for nodes in a hypercube. A simple numbering scheme can be derived from the construction of a hypercube. As illustrated in the figure, if we have a numbering of two sub-cubes of  $p/2$  nodes, we can derive a numbering scheme for the cube of  $p$  nodes by prefixing the labels of one of the sub-cubes with a "0" and the labels of the other sub-cube with a "1". This numbering scheme has the useful property that the minimum distance between two nodes is given by the number of bits that are different in the two labels. For example, nodes labeled 0110 and 0101 are two links apart, since they differ at two bit positions. This property is useful for deriving a number of parallel algorithms for the hypercube architecture.

**Tree-Based Networks**

A tree network is one in which there is only one path between any pair of nodes. Both linear arrays and star-connected networks are special cases of tree networks. Figure shows networks based on complete binary trees. Static tree networks have a processing element at each node of the tree (Figure (a)). Tree networks also have a dynamic counterpart. In a dynamic tree network, nodes at intermediate levels are switching nodes and the leaf nodes are processing elements (Figure (b)).

**Figure : Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.**

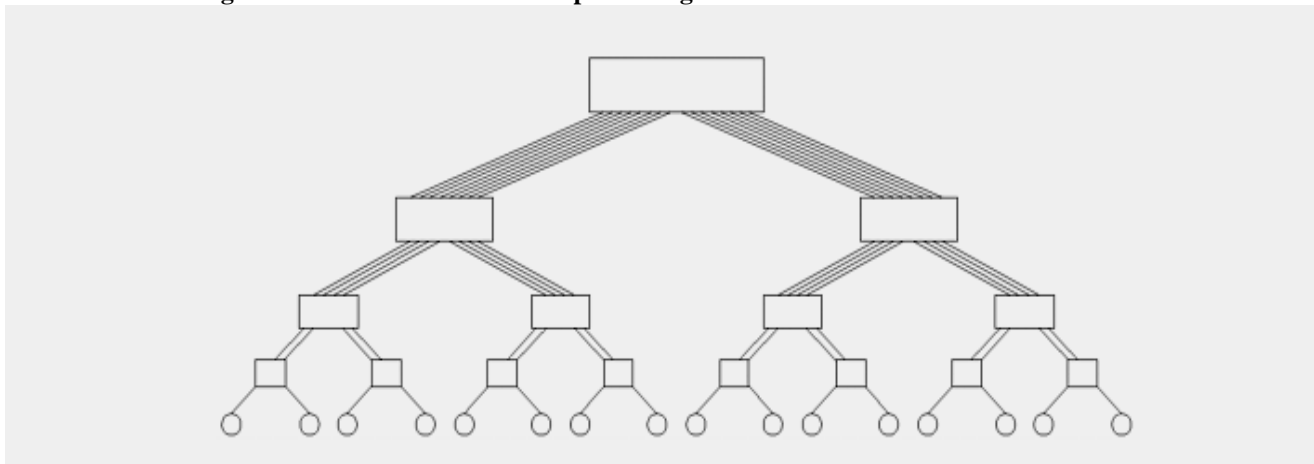




To route a message in a tree, the source node sends the message up the tree until it reaches the node at the root of the smallest sub-tree containing both the source and destination nodes. Then the message is routed down the tree towards the destination node.

Tree networks suffer from a communication bottleneck at higher levels of the tree. For example, when many nodes in the left sub-tree of a node communicate with nodes in the right sub-tree, the root node must handle all the messages. This problem can be alleviated in dynamic tree networks by increasing the number of communication links and switching nodes closer to the root. This network, also called a fat tree, is illustrated in the following figure.

Figure : A fat tree network of 16 processing nodes.



64) What are the criteria that are used to evaluate the cost and performance of static interconnection networks?

**Answer:** The various criteria used to characterize the cost and performance of static interconnection networks are:

**Diameter:** The diameter of a network is the maximum distance between any two processing nodes in the network. The distance between two processing nodes is defined as the shortest path (in terms of number of links) between them. The diameter of a completely-connected network is one, and that of a star-connected network is two. The diameter of a ring network is  $\lfloor p/2 \rfloor$ . The diameter of a two-dimensional mesh without wraparound connections is  $2(\sqrt{p} - 1)$  for the two nodes at diagonally opposed corners, and that of a wraparound mesh is  $2\lfloor \sqrt{p}/2 \rfloor$ . The diameter of a hypercube-connected network is  $\log p$  since two node labels can differ in at most  $\log p$  positions. The diameter of a complete binary tree is  $2 \log((p + 1)/2)$  because the two communicating nodes may be in separate sub-trees of the root node, and a message might have to travel all the way to the root and then down the other sub-tree.

**Connectivity:** The connectivity of a network is a measure of the multiplicity of paths between any two processing nodes. A network with high connectivity is desirable, because it lowers contention for

communication resources. One measure of connectivity is the minimum number of arcs that must be removed from the network to break it into two disconnected networks. This is called the arc connectivity of the network. The arc connectivity is one for linear arrays, as well as tree and star networks. It is two for rings and 2-D meshes without wraparound, four for 2-D wraparound meshes, and  $d$  for  $d$ -dimensional hypercubes.

**Bisection Width and Bisection Bandwidth:** The bisection width of a network is defined as the minimum number of communication links that must be removed to partition the network into two equal halves. The bisection width of a ring is two, since any partition cuts across only two communication links. Similarly, the bisection width of a two-dimensional  $p$ -node mesh without wraparound connections is  $\sqrt{p}$  and with wraparound connections is  $2\sqrt{p}$ . The bisection width of a tree and a star is one, and that of a completely-connected network of  $p$  nodes is  $p^2/4$ . The bisection width of a hypercube can be derived from its construction. We construct a  $d$ -dimensional hypercube by connecting corresponding links of two  $(d - 1)$ -dimensional hypercubes. Since each of these sub-cubes contains  $2^{(d-1)}$  or  $p/2$  nodes, at least  $p/2$  communication links must cross any partition of a hypercube into two sub-cubes.

The number of bits that can be communicated simultaneously over a link connecting two nodes is called the channel width. Channel width is equal to the number of physical wires in each communication link. The peak rate at which a single physical wire can deliver bits is called the channel rate. The peak rate at which data can be communicated between the ends of a communication link is called channel bandwidth. Channel bandwidth is the product of channel rate and channel width.

**Table : A summary of the characteristics of various static network topologies connecting  $p$  nodes.**

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p - 1$	$p(p - 1)/2$
Star	2	1	1	$p - 1$
Complete binary tree	$2 \log((p + 1)/2)$	1	1	$p - 1$
Linear array	$p - 1$	1	1	$p - 1$
2-D mesh, no wraparound	$2(\sqrt{p} - 1)$	$\sqrt{p}$	2	$2(p - \sqrt{p})$
2-D wraparound mesh	$2\lceil \sqrt{p} / 2 \rceil$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\text{Log} p$	$(p \log p)/2$
Wraparound $k$ -ary $d$ -cube	$d\lceil k / 2 \rceil$	$2k^{d-1}$	$2d$	$dp$

The **bisection bandwidth** of a network is defined as the minimum volume of communication allowed between any two halves of the network. It is the product of the bisection width and the channel bandwidth. Bisection bandwidth of a network is also sometimes referred to as cross-section bandwidth.

**Cost:** Many criteria can be used to evaluate the cost of a network. One way of defining the cost of a network is in terms of the number of communication links or the number of wires required by the network. Linear arrays and trees use only  $p - 1$  links to connect  $p$  nodes. A  $d$ -dimensional wraparound mesh has  $dp$  links. A hypercube-connected network has  $(p \log p)/2$  links.

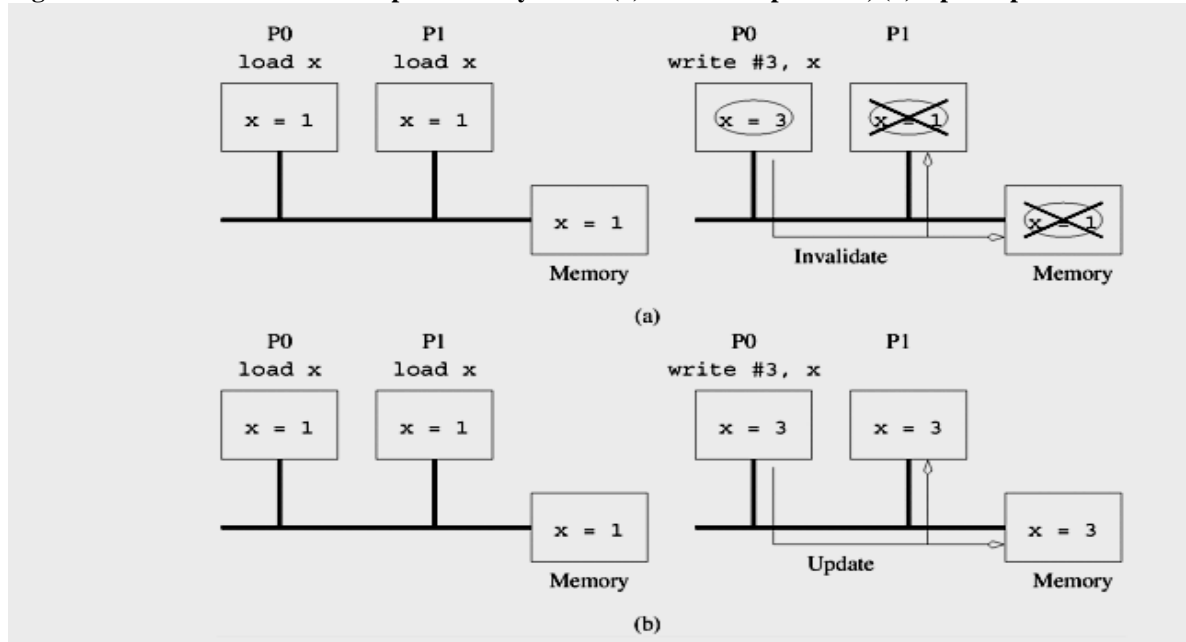
The bisection bandwidth of a network can also be used as a measure of its cost, as it provides a lower bound on the area in a two-dimensional packaging or the volume in a three-dimensional packaging. If the bisection width of a network is  $w$ , the lower bound on the area in a two-dimensional packaging is  $\Theta(w^2)$ , and the lower bound on the volume in a three-dimensional packaging is  $\Theta(w^{3/2})$ . According to this criterion, hypercubes and completely connected networks are more expensive than the other networks.

65) Explain the Cache coherence in multiprocessor system.

**Answer:** While interconnection networks provide basic mechanisms for communicating messages (data), in the case of shared-address-space computers additional hardware is required to keep multiple copies of data consistent with each other. Specifically, if there exist two copies of the data (in different caches/memory elements), how do we ensure that different processors operate on these in a manner that follows predefined semantics?

The problem of keeping caches in multiprocessor systems coherent is significantly more complex than in uniprocessor systems. This is because in addition to multiple copies as in uniprocessor systems, there may also be multiple processors modifying these copies. Consider a simple scenario illustrated in the figure. Two processors  $P_0$  and  $P_1$  are connected over a shared bus to a globally accessible memory. Both processors load the same variable. There are now three copies of the variable. The coherence mechanism must now ensure that all operations performed on these copies are serializable (i.e., there exists some serial order of instruction execution that corresponds to the parallel schedule). When a processor changes the value of its copy of the variable, one of two things must happen: the other copies must be invalidated, or the other copies must be updated. Failing this, other processors may potentially work with incorrect (stale) values of the variable. These two protocols are referred to as invalidate and update protocols and are illustrated in Figure (a) and (b).

Figure . Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.



In an update protocol, whenever a data item is written, all of its copies in the system are updated. For this reason, if a processor simply reads a data item once and never uses it, subsequent updates to this item at other processors cause excess overhead in terms of latency at source and bandwidth on the network. On the other hand, in this situation, an invalidate protocol invalidates the data item on the first update at a remote processor and subsequent updates need not be performed on this copy.

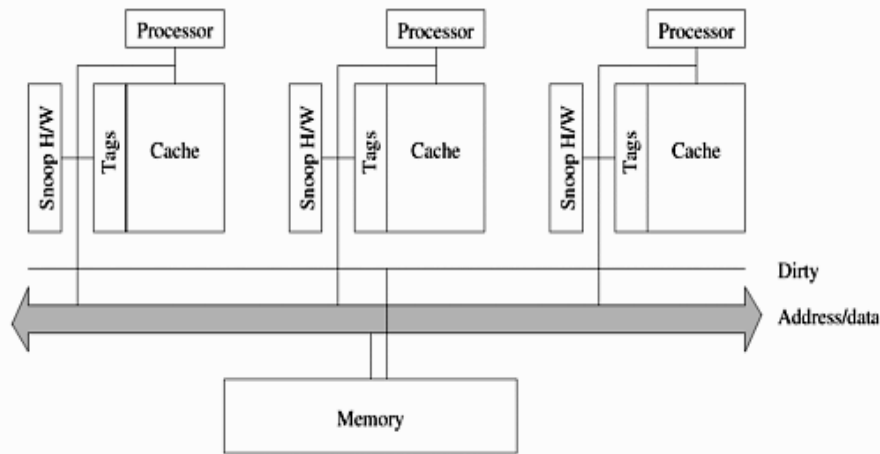
Another important factor affecting the performance of these protocols is false sharing. False sharing refers to the situation in which different processors update different parts of of the same cache-line. Thus, although the updates are not performed on shared variables, the system does not detect this. In an invalidate protocol, when a processor updates its part of the cache-line, the other copies of this line are invalidated. When other processors try to update their parts of the cache-line, the line must actually be fetched from the remote processor. It is easy to see that false-sharing can cause a cache-line to be ping-ponged between various processors. In an update protocol, this situation is slightly better since all reads

can be performed locally and the writes must be updated. This saves an invalidate operation that is otherwise wasted.

**66) Explain the Snoopy cache system**

**Answer:** Snoopy caches are typically associated with multiprocessor systems based on broadcast interconnection networks such as a bus or a ring. In such systems, all processors snoop on (monitor) the bus for transactions. This allows the processor to make state transitions for its cache-blocks. The figure illustrates a typical snoopy bus based system. Each processor's cache has a set of tag bits associated with it that determine the state of the cache blocks. These tags are updated according to the state diagram associated with the coherence protocol. For instance, when the snoop hardware detects that a read has been issued to a cache block that it has a dirty copy of, it asserts control of the bus and puts the data out. Similarly, when the snoop hardware detects that a write operation has been issued on a cache block that it has a copy of, it invalidates the block. Other state transitions are made in this fashion locally.

**Figure . A simple snoopy bus based cache coherence system.**



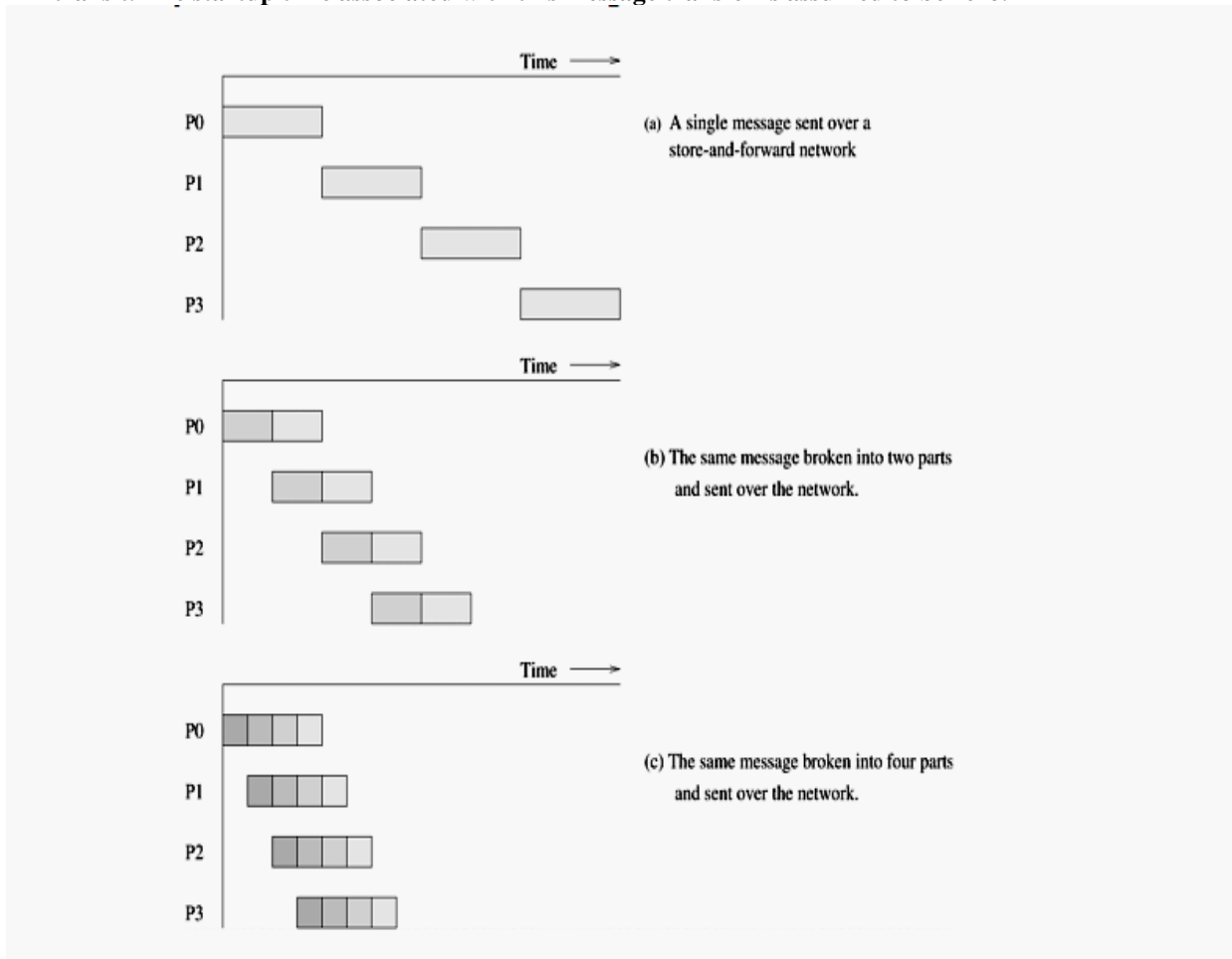
Performance of Snoopy Caches Snoopy protocols have been extensively studied and used in commercial systems. This is largely because of their simplicity and the fact that existing bus based systems can be upgraded to accommodate snoopy protocols. The performance gains of snoopy systems are derived from the fact that if different processors operate on different data items, these items can be cached. Once these items are tagged dirty, all subsequent operations can be performed locally on the cache without generating external traffic. Similarly, if a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local. In both cases, the coherence protocol does not add any overhead. On the other hand, if multiple processors read and update the same data item, they generate coherence functions across processors. Since a shared bus has a finite bandwidth, only a constant number of such coherence operations can execute in unit time. This presents a fundamental bottleneck for snoopy bus based systems.

Snoopy protocols are intimately tied to multicomputers based on broadcast networks such as buses. This is because all processors must snoop all the messages. Clearly, broadcasting all of a processor's memory operations to all the processors is not a scalable solution. An obvious solution to this problem is to propagate coherence operations only to those processors that must participate in the operation (i.e., processors that have relevant copies of the data). This solution requires us to keep track of which processors have copies of various data items and also the relevant state information for these data items. This information is stored in a directory, and the coherence mechanism based on such information is called a directory-based system.

67) Explain store-and-forward and cut-through routing.

**Answer:** In store-and-forward routing, when a message is traversing a path with multiple links, each intermediate node on the path forwards the message to the next node after it has received and stored the entire message. Figure (a) shows the communication of a message through a store-and-forward network.

Figure . Passing a message from node  $P_0$  to  $P_3$  (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.



Suppose that a message of size  $m$  is being transmitted through such a network. Assume that it traverses  $l$  links. At each link, the message incurs a cost  $t_h$  for the header and  $t_w m$  for the rest of the message to traverse the link. Since there are  $l$  such links, the total time is  $(t_h + t_w m)l$ . Therefore, for store-and-forward routing, the total communication cost for a message of size  $m$  words to traverse  $l$  communication links is

$$t_{\text{comm}} = t_s + (t_h + t_w m)l \tag{1}$$

In current parallel computers, the per-hop time  $t_h$  is quite small. For most parallel algorithms, it is less than  $t_w m$  even for small values of  $m$  and thus can be ignored. For parallel platforms using store-and-forward routing, the time given by (1) can be simplified to

$$t_{\text{comm}} = t_s + t_w m l$$

**Packet Routing**

Store-and-forward routing makes poor use of communication resources. A message is sent from one node to the next only after the entire message has been received (Figure (a)). Consider the scenario shown in Figure (b), in which the original message is broken into two equal sized parts

before it is sent. In this case, an intermediate node waits for only half of the original message to arrive before passing it on. The increased utilization of communication resources and reduced communication time is apparent from Figure (b). Figure (c) goes a step further and breaks the message into four parts. In addition to better utilization of communication resources, this principle offers other advantages – lower overhead from packet loss (errors), possibility of packets taking different paths, and better error correction capability. For these reasons, this technique is the basis for long-haul communication networks such as the Internet, where error rates, number of hops, and variation in network state can be higher. Of course, the overhead here is that each packet must carry routing, error correction, and sequencing information.

Consider the transfer of an  $m$  word message through the network. The time taken for programming the network interfaces and computing the routing information, etc., is independent of the message length. This is aggregated into the startup time  $t_s$  of the message transfer. We assume a scenario in which routing tables are static over the time of message transfer (i.e., all packets traverse the same path). While this is not a valid assumption under all circumstances, it serves the purpose of motivating a cost model for message transfer. The message is broken into packets, and packets are assembled with their error, routing, and sequencing fields. The size of a packet is now given by  $r + s$ , where  $r$  is the original message and  $s$  is the additional information carried in the packet. The time for packetizing the message is proportional to the length of the message. We denote this time by  $mt_{w1}$ . If the network is capable of communicating one word every  $t_{w2}$  seconds, incurs a delay of  $t_h$  on each hop, and if the first packet traverses  $l$  hops, then this packet takes time  $t_h l + t_{w2}(r + s)$  to reach the destination. After this time, the destination node receives an additional packet every  $t_{w2}(r + s)$  seconds. Since there are  $m/r - 1$  additional packets, the total communication time is given by:

$$\begin{aligned} t_{\text{comm}} &= t_s + mt_{w1} + t_h l + t_{w2}(r + s) + (m/r - 1) t_{w2}(r + s) \\ &= t_s + mt_{w1} + t_h l + t_{w2}m + (s/r)m t_{w2} \\ &= t_s + t_h l + m t_w \end{aligned}$$

Where

$$t_w = mt_{w1} + t_{w2}m + (s/r)m t_{w2}$$

Packet routing is suited to networks with highly dynamic states and higher error rates, such as local- and wide-area networks. This is because individual packets may take different routes and retransmissions can be localized to lost packets.

### Cut-Through Routing

In interconnection networks for parallel computers, additional restrictions can be imposed on message transfers to further reduce the overheads associated with packet switching. By forcing all packets to take the same path, we can eliminate the overhead of transmitting routing information with each packet. By forcing in-sequence delivery, sequencing information can be eliminated. By associating error information at message level rather than packet level, the overhead associated with error detection and correction can be reduced. Finally, since error rates in interconnection networks for parallel machines are extremely low, lean error detection mechanisms can be used instead of expensive error correction schemes.

The routing scheme resulting from these optimizations is called cut-through routing. In cut-through routing, a message is broken into fixed size units called flow control digits or flits. Since flits do not contain the overheads of packets, they can be much smaller than packets. A tracer is first sent from the source to the destination node to establish a connection. Once a connection has been established, the flits are sent one after the other. All flits follow the same path in a dovetailed fashion. An intermediate node does not wait for the entire message to arrive before forwarding it. As soon as a flit is received at an intermediate node, the flit is passed on to the next node. Unlike store-and-forward routing, it is no longer necessary to have buffer space at each intermediate node

to store the entire message. Therefore, cut-through routing uses less memory and memory bandwidth at intermediate nodes, and is faster.

Consider a message that is traversing such a network. If the message traverses  $l$  links, and  $t_h$  is the per-hop time, then the header of the message takes time  $lt_h$  to reach the destination. If the message is  $m$  words long, then the entire message arrives in time  $t_w m$  after the arrival of the header of the message. Therefore, the total communication time for cut-through routing is

$$t_{\text{comm}} = t_s + lt_h + t_w m$$

This time is an improvement over store-and-forward routing since terms corresponding to number of hops and number of words are additive as opposed to multiplicative in the former. Note that if the communication is between nearest neighbors (that is,  $l = 1$ ), or if the message size is small, then the communication time is similar for store-and-forward and cut-through routing schemes.

**68) Discuss the process of embedding a linear array into a hypercube?**

**Answer:** A linear array (or a ring) composed of  $2^d$  nodes (labeled 0 through  $2^d - 1$ ) can be embedded into a  $d$ -dimensional hypercube by mapping node  $i$  of the linear array onto node  $G(i, d)$  of the hypercube. The function  $G(i, x)$  is defined as follows:

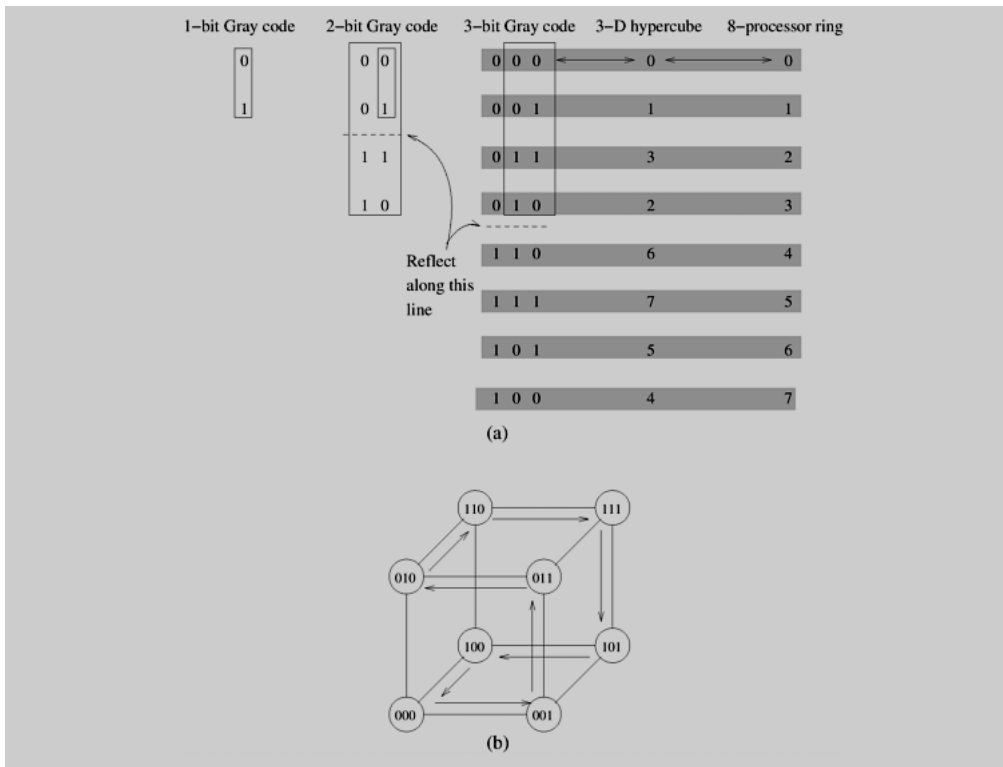
$$G(0, 1) = 0$$

$$G(1, 1) = 1$$

$$G(i, x + 1) = \begin{cases} G(i, x), & i < 2^x \\ 2^x + G(2^x + 1 - i, x), & i \geq 2^x \end{cases}$$

The function  $G$  is called the binary reflected Gray code (RGC). The entry  $G(i, d)$  denotes the  $i$ th entry in the sequence of Gray codes of  $d$  bits. Gray codes of  $d + 1$  bits are derived from a table of Gray codes of  $d$  bits by reflecting the table and prefixing the reflected entries with a 1 and the original entries with a 0.

**Figure . (a) A three-bit reflected Gray code ring; and (b) its embedding into a three-dimensional hypercube.**



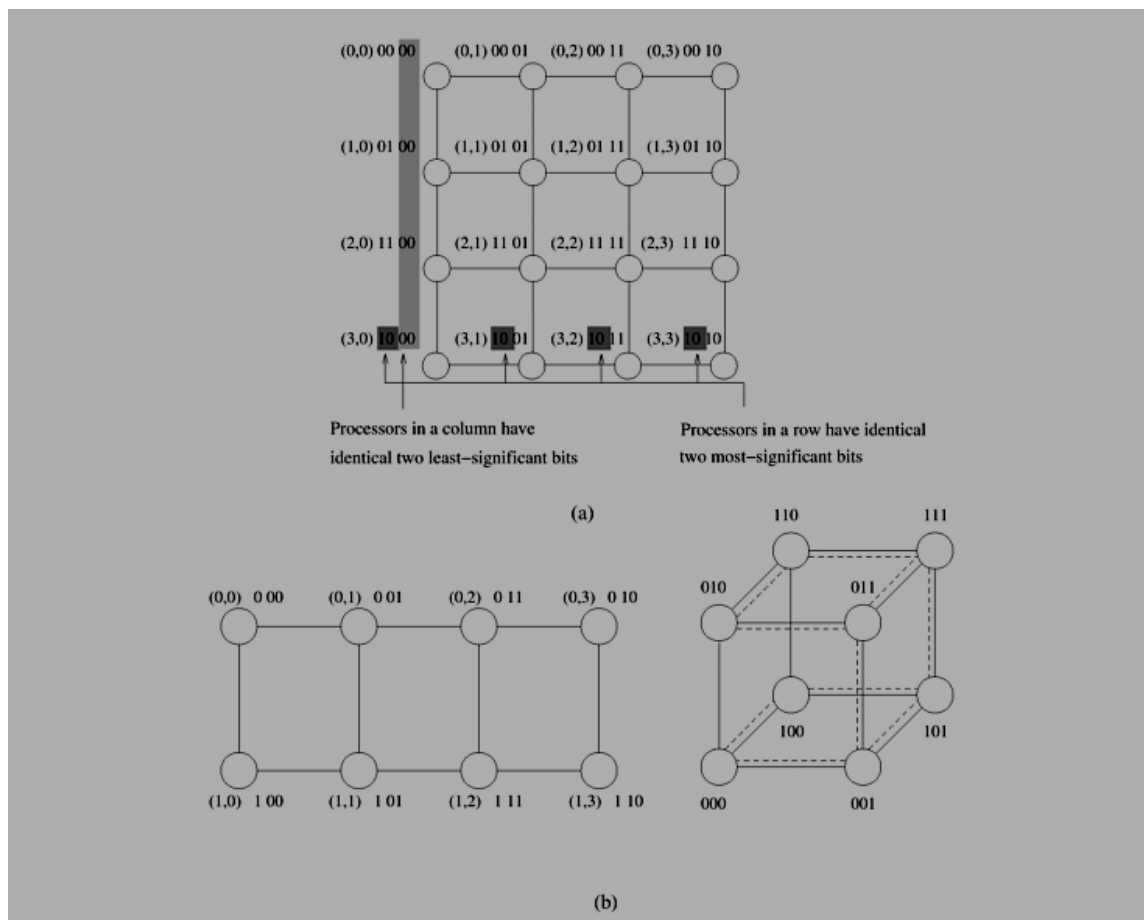
A careful look at the Gray code table reveals that two adjoining entries ( $G(i, d)$  and  $G(i + 1, d)$ ) differ from each other at only one bit position. Since node  $i$  in the linear array is mapped to node  $G(i, d)$ , and node  $i + 1$  is mapped to  $G(i + 1, d)$ , there is a direct link in the hypercube that corresponds to each direct link in the linear array. (Recall that two nodes whose labels differ at only one bit position have a direct link in a hypercube.) Therefore, the mapping specified by the function  $G$  has a dilation of one and a congestion of one.

**69) Discuss the process of embedding a Mesh into a hypercube?**

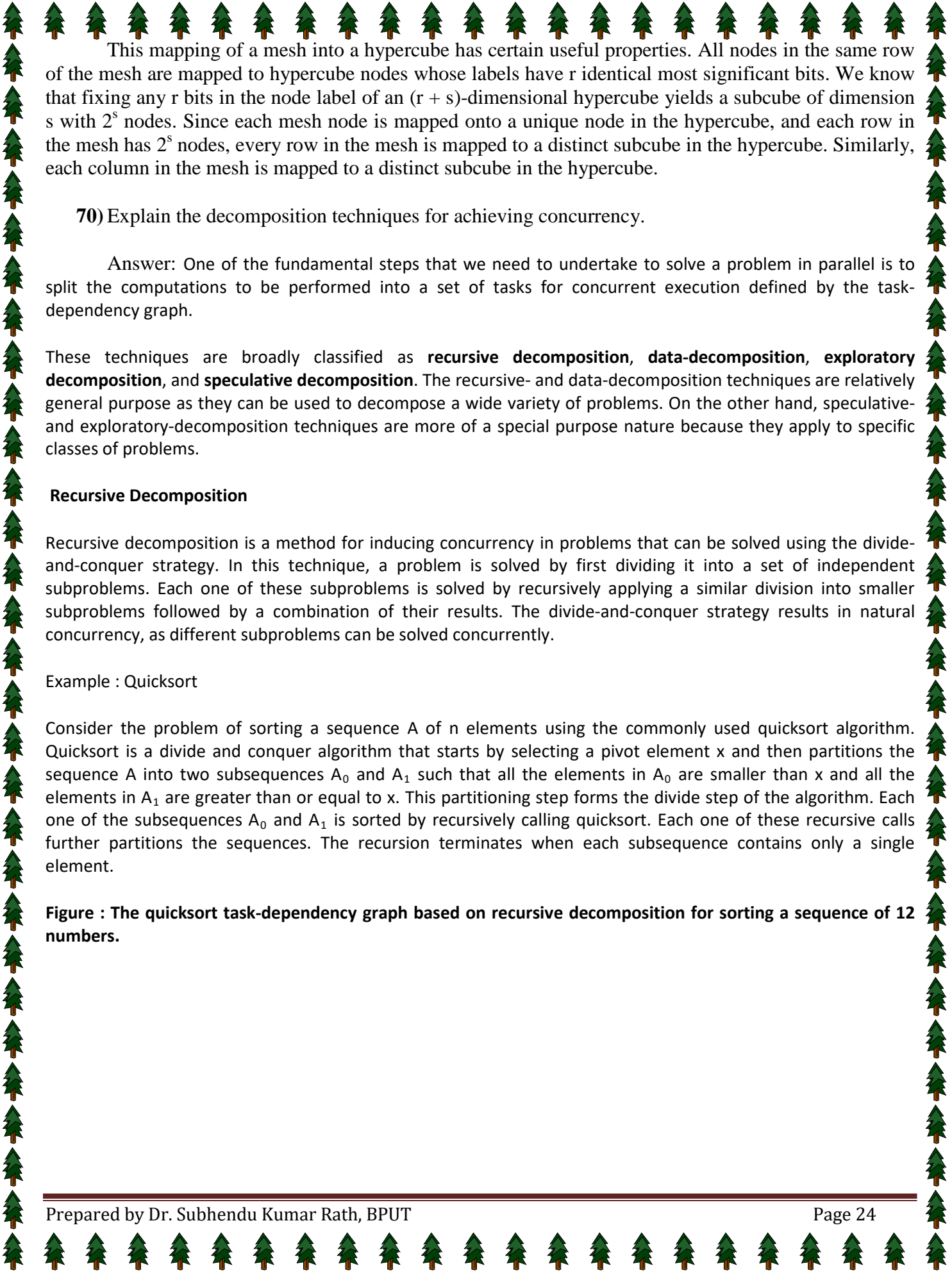
Answer: Embedding a mesh into a hypercube is a natural extension of embedding a ring into a hypercube. We can embed a  $2^r \times 2^s$  wraparound mesh into a  $2^{r+s}$ -node hypercube by mapping node  $(i, j)$  of the mesh onto node  $G(i, r - 1) || G(j, s - 1)$  of the hypercube (where  $||$  denotes concatenation of the two Gray codes). Note that immediate neighbors in the mesh are mapped to hypercube nodes whose labels differ in exactly one bit position. Therefore, this mapping has a dilation of one and a congestion of one.

For example, consider embedding a  $2 \times 4$  mesh into an eight-node hypercube. The values of  $r$  and  $s$  are 1 and 2, respectively. Node  $(i, j)$  of the mesh is mapped to node  $G(i, 1) || G(j, 2)$  of the hypercube. Therefore, node  $(0, 0)$  of the mesh is mapped to node 000 of the hypercube, because  $G(0, 1)$  is 0 and  $G(0, 2)$  is 00; concatenating the two yields the label 000 for the hypercube node. Similarly, node  $(0, 1)$  of the mesh is mapped to node 001 of the hypercube, and so on.

Figure (a) A  $4 \times 4$  mesh illustrating the mapping of mesh nodes to the nodes in a four-dimensional hypercube; and (b) a  $2 \times 4$  mesh embedded into a three-dimensional hypercube.







This mapping of a mesh into a hypercube has certain useful properties. All nodes in the same row of the mesh are mapped to hypercube nodes whose labels have  $r$  identical most significant bits. We know that fixing any  $r$  bits in the node label of an  $(r + s)$ -dimensional hypercube yields a subcube of dimension  $s$  with  $2^s$  nodes. Since each mesh node is mapped onto a unique node in the hypercube, and each row in the mesh has  $2^s$  nodes, every row in the mesh is mapped to a distinct subcube in the hypercube. Similarly, each column in the mesh is mapped to a distinct subcube in the hypercube.

70) Explain the decomposition techniques for achieving concurrency.

Answer: One of the fundamental steps that we need to undertake to solve a problem in parallel is to split the computations to be performed into a set of tasks for concurrent execution defined by the task-dependency graph.

These techniques are broadly classified as **recursive decomposition**, **data-decomposition**, **exploratory decomposition**, and **speculative decomposition**. The recursive- and data-decomposition techniques are relatively general purpose as they can be used to decompose a wide variety of problems. On the other hand, speculative- and exploratory-decomposition techniques are more of a special purpose nature because they apply to specific classes of problems.

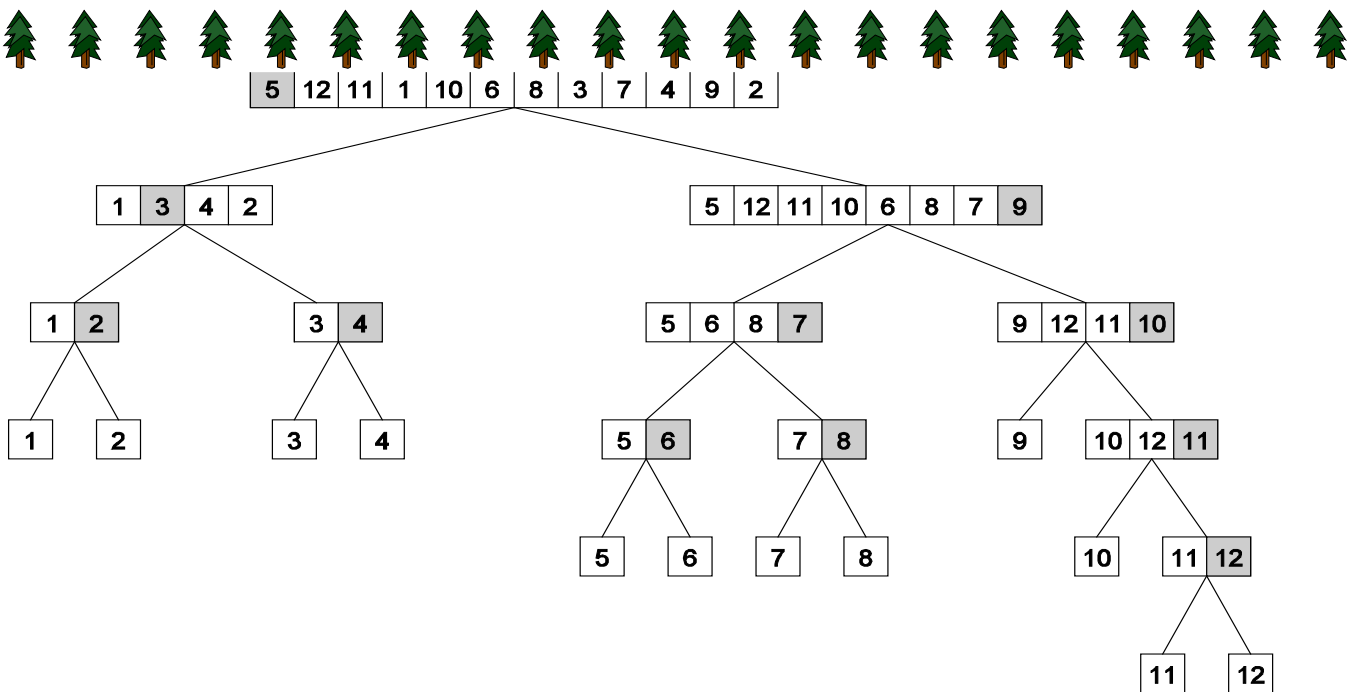
### Recursive Decomposition

Recursive decomposition is a method for inducing concurrency in problems that can be solved using the divide-and-conquer strategy. In this technique, a problem is solved by first dividing it into a set of independent subproblems. Each one of these subproblems is solved by recursively applying a similar division into smaller subproblems followed by a combination of their results. The divide-and-conquer strategy results in natural concurrency, as different subproblems can be solved concurrently.

Example : Quicksort

Consider the problem of sorting a sequence  $A$  of  $n$  elements using the commonly used quicksort algorithm. Quicksort is a divide and conquer algorithm that starts by selecting a pivot element  $x$  and then partitions the sequence  $A$  into two subsequences  $A_0$  and  $A_1$  such that all the elements in  $A_0$  are smaller than  $x$  and all the elements in  $A_1$  are greater than or equal to  $x$ . This partitioning step forms the divide step of the algorithm. Each one of the subsequences  $A_0$  and  $A_1$  is sorted by recursively calling quicksort. Each one of these recursive calls further partitions the sequences. The recursion terminates when each subsequence contains only a single element.

Figure : The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.



In Figure, we define a task as the work of partitioning a given subsequence. Initially, there is only one sequence (i.e., the root of the tree), and we can use only a single process to partition it. The completion of the root task results in two subsequences ( $A_0$  and  $A_1$ , corresponding to the two nodes at the first level of the tree) and each one can be partitioned in parallel. Similarly, the concurrency continues to increase as we move down the tree.

**Data Decomposition**

Data decomposition is a powerful and commonly used method for deriving concurrency in algorithms that operate on large data structures. In this method, the decomposition of computations is done in two steps.

In the first step, the data on which the computations are performed is partitioned, and in the second step, this data partitioning is used to induce a partitioning of the computations into tasks. The operations that these tasks perform on different data partitions are usually similar (e.g., matrix multiplication) or are chosen from a small set of operations (e.g., LU factorization)

The partitioning of data can be performed in many possible ways as discussed next. In general, one must explore and evaluate all possible ways of partitioning the data and determine which one yields a natural and efficient computational decomposition.

**Partitioning Output Data :** In many computations, each element of the output can be computed independently of others as a function of the input. In such computations, a partitioning of the output data automatically induces a decomposition of the problems into tasks, where each task is assigned the work of computing a portion of the output.

Example : Matrix multiplication

Consider the problem of multiplying two  $n \times n$  matrices A and B to yield a matrix C. Figure shows a decomposition of this problem into four tasks. Each matrix is considered to be composed of four blocks or submatrices defined by splitting each dimension of the matrix into half. The four submatrices of C, roughly of size  $n/2 \times n/2$  each, are then independently computed by four tasks as the sums of the appropriate products of submatrices of A and B.

Figure : (a) Partitioning of input and output matrices into 2 x 2 submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a).

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Most matrix algorithms, including matrix-vector and matrix-matrix multiplication, can be formulated in terms of block matrix operations. In such a formulation, the matrix is viewed as composed of blocks or submatrices and the scalar arithmetic operations on its elements are replaced by the equivalent matrix operations on the blocks.

**Partitioning Input Data** : Partitioning of output data can be performed only if each output can be naturally computed as a function of the input. In many algorithms, it is not possible or desirable to partition the output data. For example, while finding the minimum, maximum, or the sum of a set of numbers, the output is a single unknown value. In a sorting algorithm, the individual elements of the output cannot be efficiently determined in isolation. In such cases, it is sometimes possible to partition the input data, and then use this partitioning to induce concurrency. A task is created for each partition of the input data and this task performs as much computation as possible using these local data. Note that the solutions to tasks induced by input partitions may not directly solve the original problem. In such cases, a follow-up computation is needed to combine the results. For example, while finding the sum of a sequence of N numbers using p processes (N > p), we can partition the input into p subsets of nearly equal sizes. Each task then computes the sum of the numbers in one of the subsets. Finally, the p partial results can be added up to yield the final result.

**Partitioning both Input and Output Data** : In some cases, in which it is possible to partition the output data, partitioning of input data can offer additional concurrency.

**Partitioning Intermediate Data** : Algorithms are often structured as multi-stage computations such that the output of one stage is the input to the subsequent stage. A decomposition of such an algorithm can be derived by partitioning the input or the output data of an intermediate stage of the algorithm. Partitioning intermediate data can sometimes lead to higher concurrency than partitioning input or output data. Often, the intermediate data are not generated explicitly in the serial algorithm for solving the problem and some restructuring of the original algorithm may be required to use intermediate data partitioning to induce a decomposition.

**The Owner-Computes Rule** : A decomposition based on partitioning output or input data is also widely referred to as the **owner-computes rule**. The idea behind this rule is that each partition performs all the computations involving data that it owns. Depending on the nature of the data or the type of data-partitioning, the owner-computes rule may mean different things. For instance, when we assign partitions of the input data to tasks, then the owner-computes rule means that a task performs all the computations that can be done using these data. On the other hand, if we partition the output data, then the owner-computes rule means that a task computes all the data in the partition assigned to it.

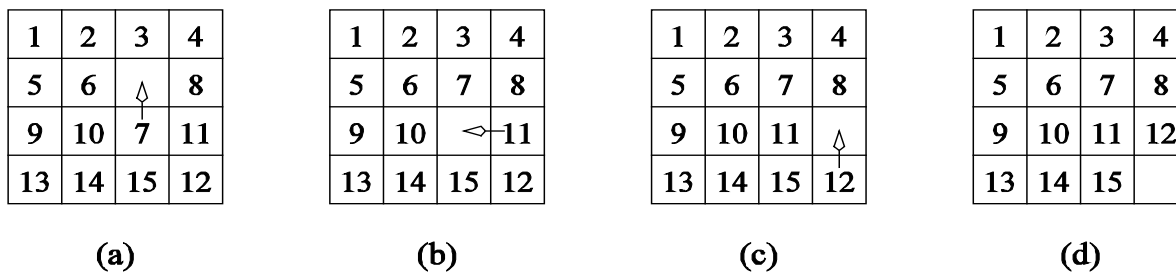
### Exploratory Decomposition

Exploratory decomposition is used to decompose problems whose underlying computations correspond to a search of a space for solutions. In exploratory decomposition, we partition the search space into smaller parts, and search each one of these parts concurrently, until the desired solutions are found. For an example of exploratory decomposition, consider the 15-puzzle problem.

Example : The 15-puzzle problem

The 15-puzzle consists of 15 tiles numbered 1 through 15 and one blank tile placed in a 4 x 4 grid. A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tile's original position. Depending on the configuration of the grid, up to four moves are possible: up, down, left, and right. The initial and final configurations of the tiles are specified. The objective is to determine any sequence or a shortest sequence of moves that transforms the initial configuration to the final configuration. The Figure illustrates sample initial and final configurations and a sequence of moves leading from the initial configuration to the final configuration.

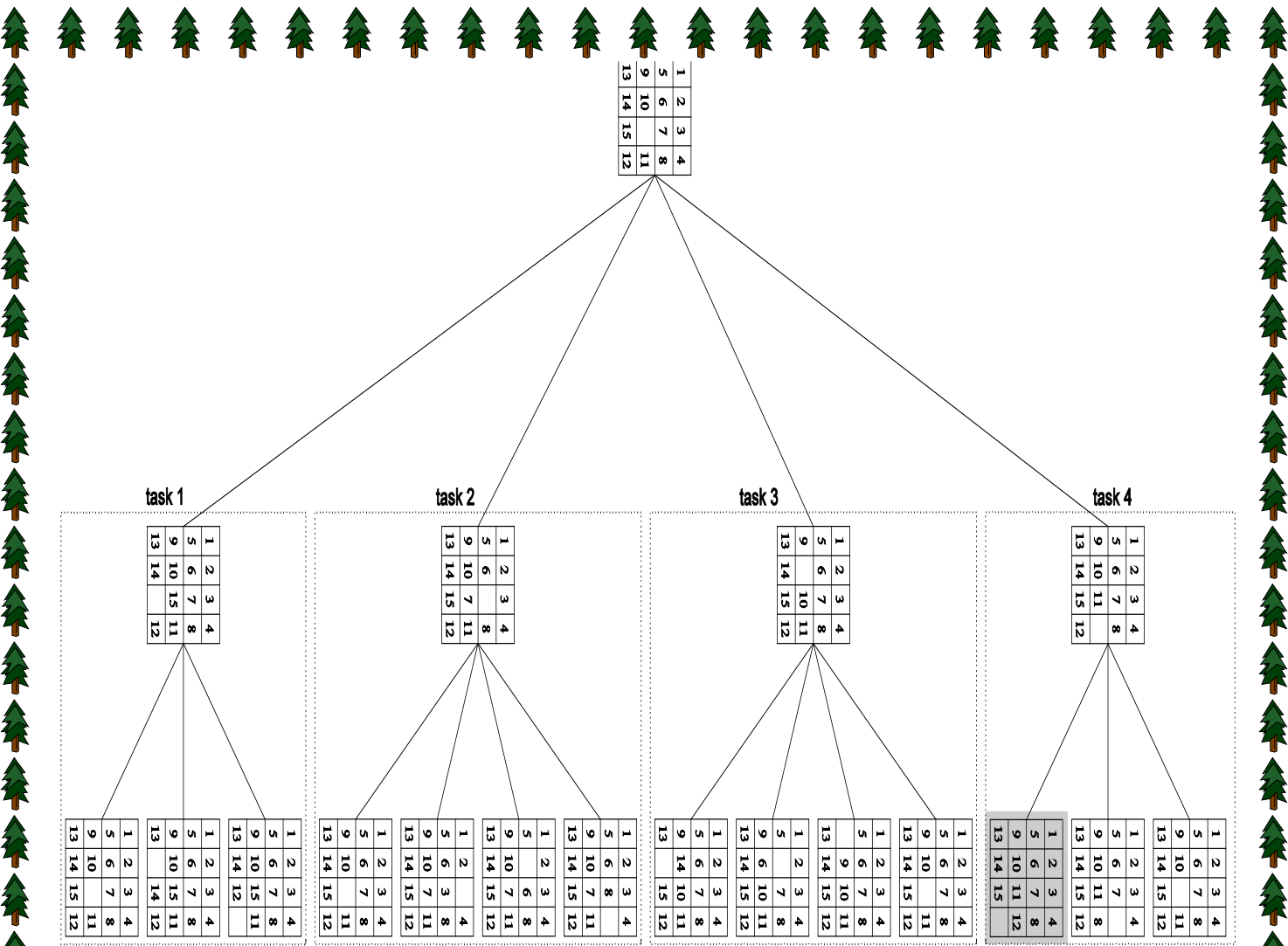
Figure. A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.



The 15-puzzle is typically solved using tree-search techniques. Starting from the initial configuration, all possible successor configurations are generated. A configuration may have 2, 3, or 4 possible successor configurations, each corresponding to the occupation of the empty slot by one of its neighbors. The task of finding a path from initial to final configuration now translates to finding a path from one of these newly generated configurations to the final configuration. Since one of these newly generated configurations must be closer to the solution by one move (if a solution exists), we have made some progress towards finding the solution. The configuration space generated by the tree search is often referred to as a state space graph. Each node of the graph is a configuration and each edge of the graph connects configurations that can be reached from one another by a single move of a tile.

One method for solving this problem in parallel is as follows. First, a few levels of configurations starting from the initial configuration are generated serially until the search tree has a sufficient number of leaf nodes (i.e., configurations of the 15-puzzle). Now each node is assigned to a task to explore further until at least one of them finds a solution. As soon as one of the concurrent tasks finds a solution it can inform the others to terminate their searches. The following figure illustrates one such decomposition into four tasks in which task 4 finds the solution.

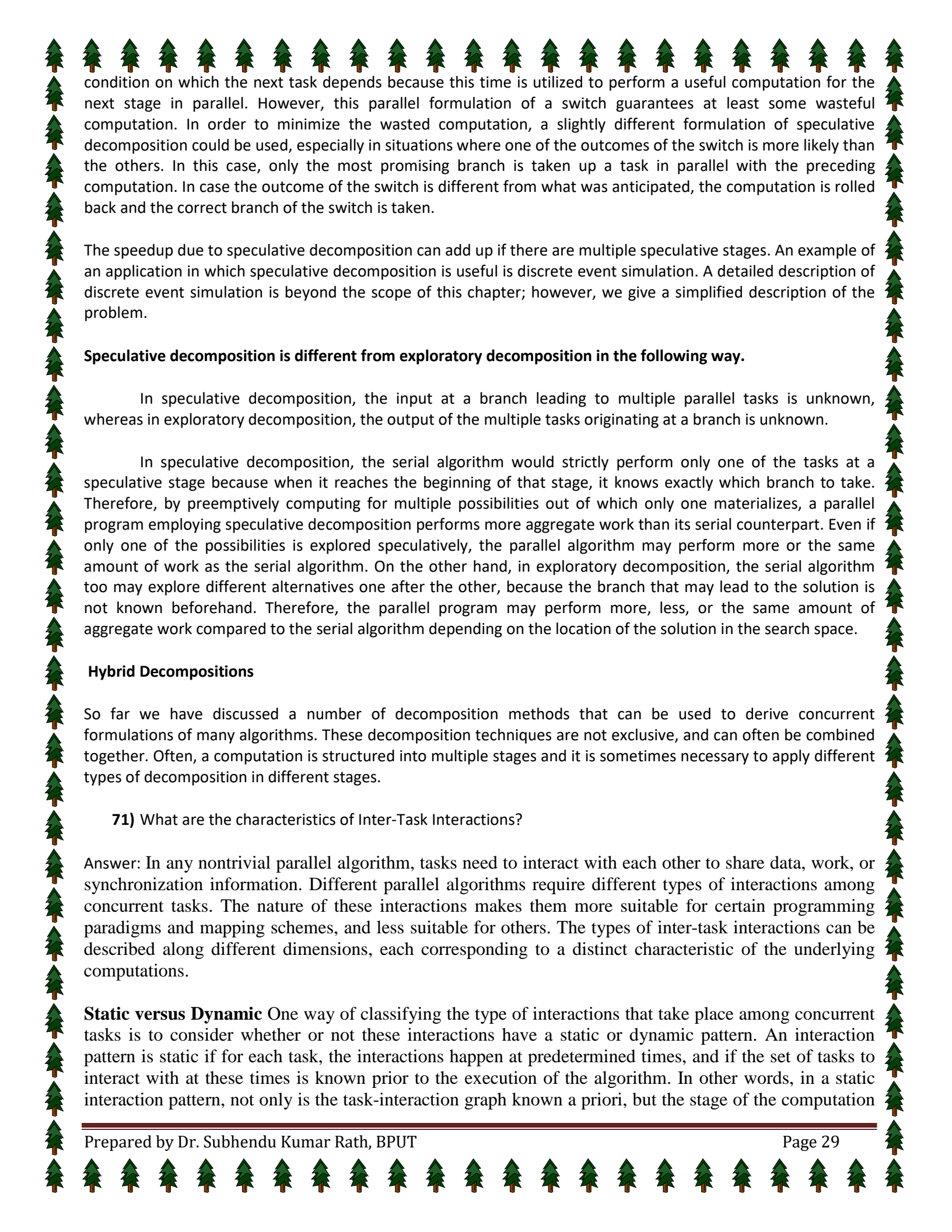
Figure : The states generated by an instance of the 15-puzzle problem.



Note that even though exploratory decomposition may appear similar to data-decomposition (the search space can be thought of as being the data that get partitioned) it is fundamentally different in the following way. The tasks induced by data-decomposition are performed in their entirety and each task performs useful computations towards the solution of the problem. On the other hand, in exploratory decomposition, unfinished tasks can be terminated as soon as an overall solution is found. Hence, the portion of the search space searched (and the aggregate amount of work performed) by a parallel formulation can be very different from that searched by a serial algorithm. The work performed by the parallel formulation can be either smaller or greater than that performed by the serial algorithm.

### Speculative Decomposition

Speculative decomposition is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. In this situation, while one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage. This scenario is similar to evaluating one or more of the branches of a switch statement in C in parallel before the input for the switch are available. While one task is performing the computation that will eventually resolve the switch, other tasks could pick up the multiple branches of the switch in parallel. When the input for the switch has finally been computed, the computation corresponding to the correct branch would be used while that corresponding to the other branches would be discarded. The parallel run time is smaller than the serial run time by the amount of time required evaluating the



condition on which the next task depends because this time is utilized to perform a useful computation for the next stage in parallel. However, this parallel formulation of a switch guarantees at least some wasteful computation. In order to minimize the wasted computation, a slightly different formulation of speculative decomposition could be used, especially in situations where one of the outcomes of the switch is more likely than the others. In this case, only the most promising branch is taken up a task in parallel with the preceding computation. In case the outcome of the switch is different from what was anticipated, the computation is rolled back and the correct branch of the switch is taken.

The speedup due to speculative decomposition can add up if there are multiple speculative stages. An example of an application in which speculative decomposition is useful is discrete event simulation. A detailed description of discrete event simulation is beyond the scope of this chapter; however, we give a simplified description of the problem.

**Speculative decomposition is different from exploratory decomposition in the following way.**

In speculative decomposition, the input at a branch leading to multiple parallel tasks is unknown, whereas in exploratory decomposition, the output of the multiple tasks originating at a branch is unknown.

In speculative decomposition, the serial algorithm would strictly perform only one of the tasks at a speculative stage because when it reaches the beginning of that stage, it knows exactly which branch to take. Therefore, by preemptively computing for multiple possibilities out of which only one materializes, a parallel program employing speculative decomposition performs more aggregate work than its serial counterpart. Even if only one of the possibilities is explored speculatively, the parallel algorithm may perform more or the same amount of work as the serial algorithm. On the other hand, in exploratory decomposition, the serial algorithm too may explore different alternatives one after the other, because the branch that may lead to the solution is not known beforehand. Therefore, the parallel program may perform more, less, or the same amount of aggregate work compared to the serial algorithm depending on the location of the solution in the search space.


**Hybrid Decompositions**

So far we have discussed a number of decomposition methods that can be used to derive concurrent formulations of many algorithms. These decomposition techniques are not exclusive, and can often be combined together. Often, a computation is structured into multiple stages and it is sometimes necessary to apply different types of decomposition in different stages.

**71) What are the characteristics of Inter-Task Interactions?**

Answer: In any nontrivial parallel algorithm, tasks need to interact with each other to share data, work, or synchronization information. Different parallel algorithms require different types of interactions among concurrent tasks. The nature of these interactions makes them more suitable for certain programming paradigms and mapping schemes, and less suitable for others. The types of inter-task interactions can be described along different dimensions, each corresponding to a distinct characteristic of the underlying computations.

**Static versus Dynamic** One way of classifying the type of interactions that take place among concurrent tasks is to consider whether or not these interactions have a static or dynamic pattern. An interaction pattern is static if for each task, the interactions happen at predetermined times, and if the set of tasks to interact with at these times is known prior to the execution of the algorithm. In other words, in a static interaction pattern, not only is the task-interaction graph known a priori, but the stage of the computation



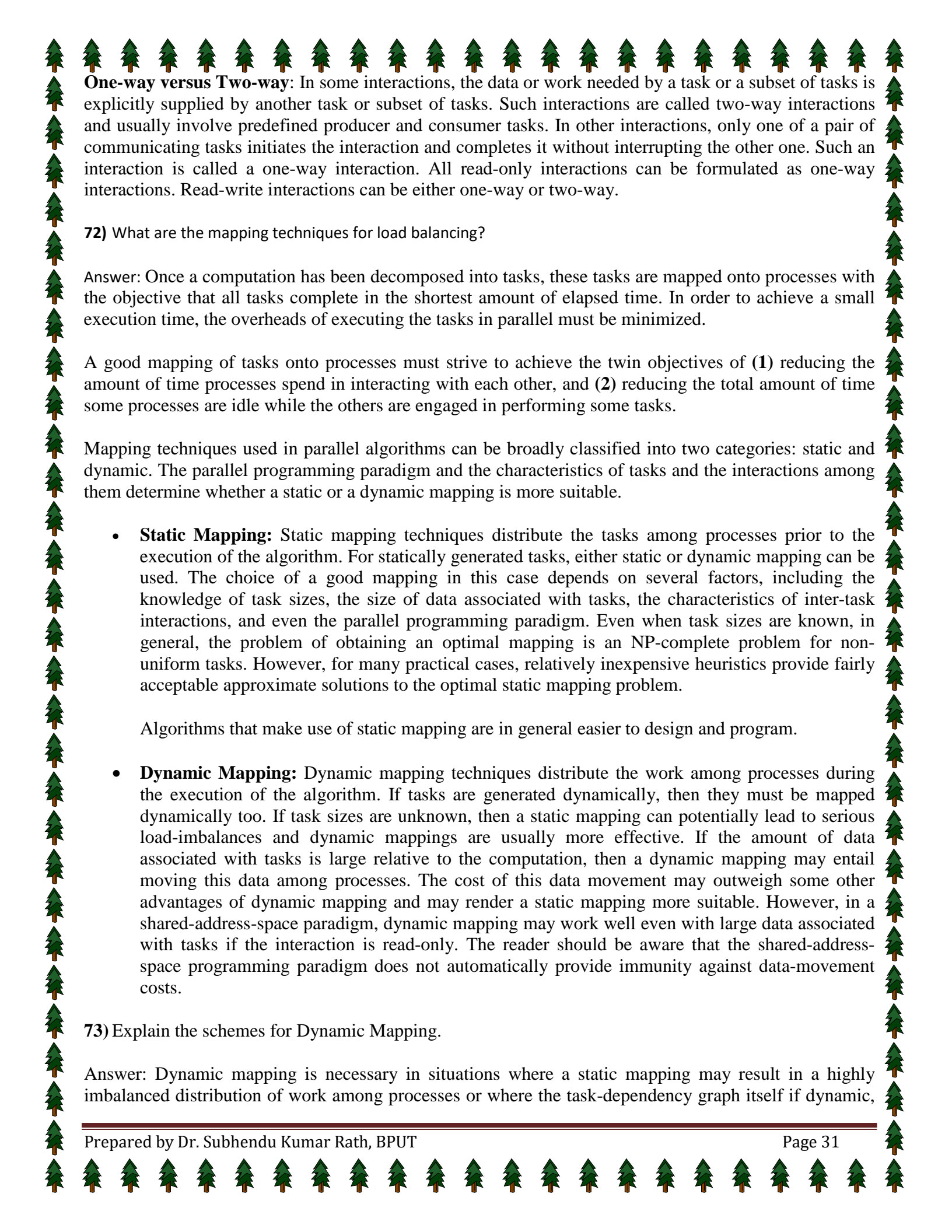
at which each interaction occurs is also known. An interaction pattern is dynamic if the timing of interactions or the set of tasks to interact with cannot be determined prior to the execution of the algorithm.

Static interactions can be programmed easily in the message-passing paradigm, but dynamic interactions are harder to program. The reason is that interactions in message-passing require active involvement of both interacting tasks – the sender and the receiver of information. The unpredictable nature of dynamic iterations makes it hard for both the sender and the receiver to participate in the interaction at the same time. Therefore, when implementing a parallel algorithm with dynamic interactions in the message-passing paradigm, the tasks must be assigned additional synchronization or polling responsibility. Shared-address space programming can code both types of interactions equally easily.

The decompositions for parallel matrix multiplication presented earlier in this chapter exhibit static inter-task interactions. For an example of dynamic interactions, consider solving the 15-puzzle problem in which tasks are assigned different states to explore after an initial step that generates the desirable number of states by applying breadth-first search on the initial state. It is possible that a certain state leads to all dead ends and a task exhausts its search space without reaching the goal state, while other tasks are still busy trying to find a solution. The task that has exhausted its work can pick up an unexplored state from the queue of another busy task and start exploring it. The interactions involved in such a transfer of work from one task to another are dynamic.

**Regular versus Irregular:** Another way of classifying the interactions is based upon their spatial structure. An interaction pattern is considered to be regular if it has some structure that can be exploited for efficient implementation. On the other hand, an interaction pattern is called irregular if no such regular pattern exists. Irregular and dynamic communications are harder to handle, particularly in the message-passing programming paradigm. An example of a decomposition with a regular interaction pattern is the problem of image dithering.

**Read-only versus Read-Write:** We have already learned that sharing of data among tasks leads to inter-task interaction. However, the type of sharing may impact the choice of the mapping. Data sharing interactions can be categorized as either read-only or read-write interactions. As the name suggests, in read-only interactions, tasks require only a read-access to the data shared among many concurrent tasks. For example, in the decomposition for parallel matrix multiplication, the tasks only need to read the shared input matrices A and B. In read-write interactions, multiple tasks need to read and write on some shared data. For example, consider the problem of solving the 15-puzzle. In this formulation, each state is considered an equally suitable candidate for further expansion. The search can be made more efficient if the states that appear to be closer to the solution are given a priority for further exploration. An alternative search technique known as heuristic search implements such a strategy. In heuristic search, we use a heuristic to provide a relative approximate indication of the distance of each state from the solution (i.e. the potential number of moves required to reach the solution). In the case of the 15-puzzle, the number of tiles that are out of place in a given state could serve as such a heuristic. The states that need to be expanded further are stored in a priority queue based on the value of this heuristic. While choosing the states to expand, we give preference to more promising states, i.e. the ones that have fewer out-of-place tiles and hence, are more likely to lead to a quick solution. In this situation, the priority queue constitutes shared data and tasks need both read and write access to it; they need to put the states resulting from an expansion into the queue and they need to pick up the next most promising state for the next expansion.



**One-way versus Two-way:** In some interactions, the data or work needed by a task or a subset of tasks is explicitly supplied by another task or subset of tasks. Such interactions are called two-way interactions and usually involve predefined producer and consumer tasks. In other interactions, only one of a pair of communicating tasks initiates the interaction and completes it without interrupting the other one. Such an interaction is called a one-way interaction. All read-only interactions can be formulated as one-way interactions. Read-write interactions can be either one-way or two-way.

72) What are the mapping techniques for load balancing?

Answer: Once a computation has been decomposed into tasks, these tasks are mapped onto processes with the objective that all tasks complete in the shortest amount of elapsed time. In order to achieve a small execution time, the overheads of executing the tasks in parallel must be minimized.

A good mapping of tasks onto processes must strive to achieve the twin objectives of (1) reducing the amount of time processes spend in interacting with each other, and (2) reducing the total amount of time some processes are idle while the others are engaged in performing some tasks.

Mapping techniques used in parallel algorithms can be broadly classified into two categories: static and dynamic. The parallel programming paradigm and the characteristics of tasks and the interactions among them determine whether a static or a dynamic mapping is more suitable.

- **Static Mapping:** Static mapping techniques distribute the tasks among processes prior to the execution of the algorithm. For statically generated tasks, either static or dynamic mapping can be used. The choice of a good mapping in this case depends on several factors, including the knowledge of task sizes, the size of data associated with tasks, the characteristics of inter-task interactions, and even the parallel programming paradigm. Even when task sizes are known, in general, the problem of obtaining an optimal mapping is an NP-complete problem for non-uniform tasks. However, for many practical cases, relatively inexpensive heuristics provide fairly acceptable approximate solutions to the optimal static mapping problem.


Algorithms that make use of static mapping are in general easier to design and program.

- **Dynamic Mapping:** Dynamic mapping techniques distribute the work among processes during the execution of the algorithm. If tasks are generated dynamically, then they must be mapped dynamically too. If task sizes are unknown, then a static mapping can potentially lead to serious load-imbalances and dynamic mappings are usually more effective. If the amount of data associated with tasks is large relative to the computation, then a dynamic mapping may entail moving this data among processes. The cost of this data movement may outweigh some other advantages of dynamic mapping and may render a static mapping more suitable. However, in a shared-address-space paradigm, dynamic mapping may work well even with large data associated with tasks if the interaction is read-only. The reader should be aware that the shared-address-space programming paradigm does not automatically provide immunity against data-movement costs.

73) Explain the schemes for Dynamic Mapping.

Answer: Dynamic mapping is necessary in situations where a static mapping may result in a highly imbalanced distribution of work among processes or where the task-dependency graph itself is dynamic,





thus precluding a static mapping. Since the primary reason for using a dynamic mapping is balancing the workload among processes, dynamic mapping is often referred to as dynamic load-balancing. Dynamic mapping techniques are usually classified as either centralized or distributed.

### Centralized Schemes

In a centralized dynamic load balancing scheme, all executable tasks are maintained in a common central data structure or they are maintained by a special process or a subset of processes. If a special process is designated to manage the pool of available tasks, then it is often referred to as the master and the other processes that depend on the master to obtain work are referred to as slaves. Whenever a process has no work, it takes a portion of available work from the central data structure or the master process. Whenever a new task is generated, it is added to this centralized data structure or reported to the master process. Centralized load-balancing schemes are usually easier to implement than distributed schemes, but may have limited scalability. As more and more processes are used, the large number of accesses to the common data structure or the master process tends to become a bottleneck.

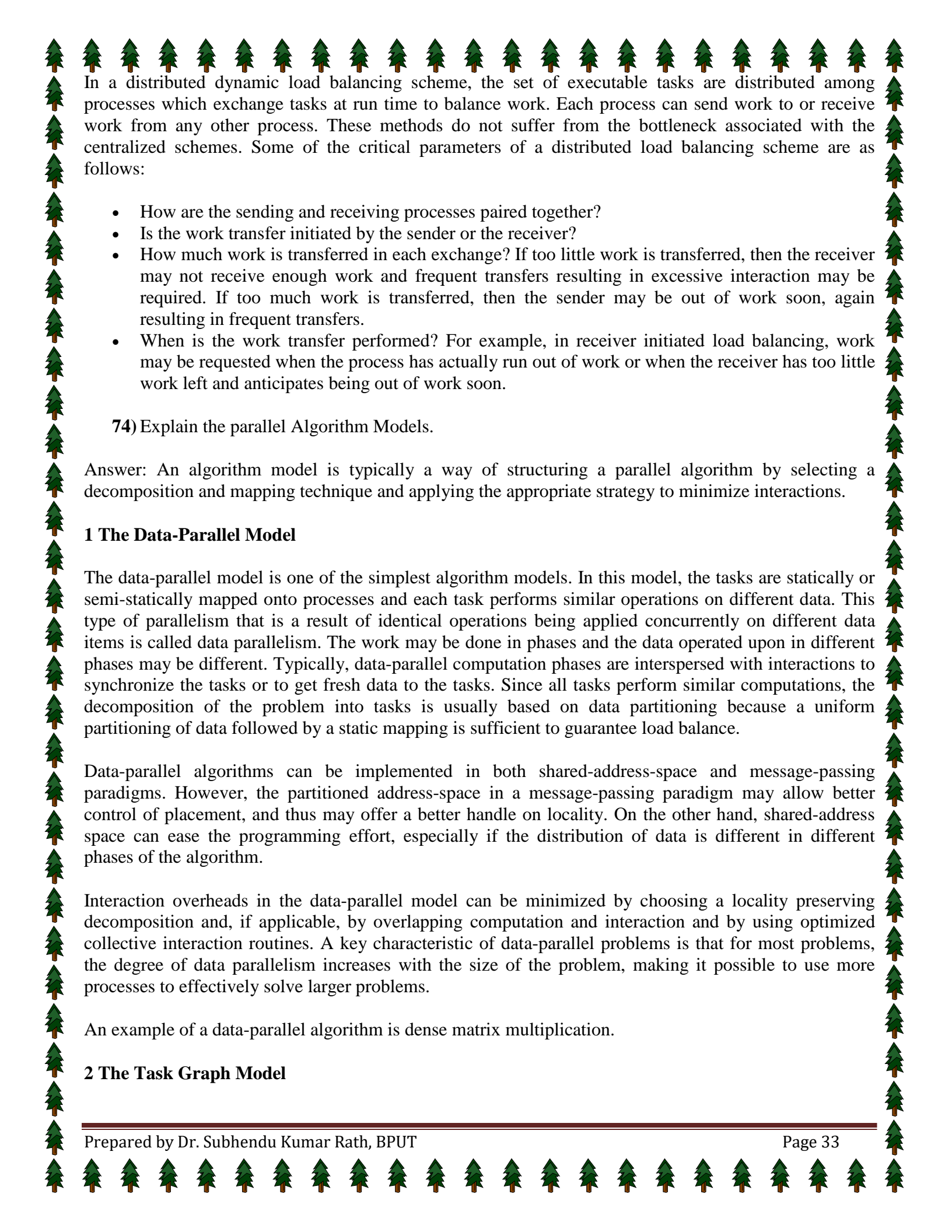
As an example of a scenario where centralized mapping may be applicable, consider the problem of sorting the entries in each row of an  $n \times n$  matrix  $A$ . Serially, this can be accomplished by the following simple program segment:

```
1  for (i=1; i<n; i++)
2      sort(A[i],n);
```

Recall that the time to sort an array using some of the commonly used sorting algorithms, such as quicksort, can vary significantly depending on the initial order of the elements to be sorted. Therefore, each iteration of the loop in the program shown above can take different amounts of time. A naive mapping of the task of sorting an equal number of rows to each process may lead to load-imbalance. A possible solution to the potential load-imbalance problem in this case would be to maintain a central pool of indices of the rows that have yet to be sorted. Whenever a process is idle, it picks up an available index, deletes it, and sorts the row with that index, as long as the pool of indices is not empty. This method of scheduling the independent iterations of a loop among parallel processes is known as self scheduling.

The assignment of a single task to a process at a time is quite effective in balancing the computation; however, it may lead to bottlenecks in accessing the shared work queue, especially if each task (i.e., each loop iteration in this case) does not require a large enough amount of computation. If the average size of each task is  $M$ , and it takes  $\Delta$  time to assign work to a process, then at most  $M/\Delta$  processes can be kept busy effectively. The bottleneck can be eased by getting more than one task at a time. In chunk scheduling, every time a process runs out of work it gets a group of tasks. The potential problem with such a scheme is that it may lead to load-imbalance if the number of tasks (i.e., chunk) assigned in a single step is large. The danger of load-imbalance due to large chunk sizes can be reduced by decreasing the chunk-size as the program progresses. That is, initially the chunk size is large, but as the number of iterations left to be executed decreases, the chunk size also decreases. A variety of schemes have been developed for gradually adjusting the chunk size, that decrease the chunk size either linearly or non-linearly.

### Distributed Schemes



In a distributed dynamic load balancing scheme, the set of executable tasks are distributed among processes which exchange tasks at run time to balance work. Each process can send work to or receive work from any other process. These methods do not suffer from the bottleneck associated with the centralized schemes. Some of the critical parameters of a distributed load balancing scheme are as follows:

- How are the sending and receiving processes paired together?
- Is the work transfer initiated by the sender or the receiver?
- How much work is transferred in each exchange? If too little work is transferred, then the receiver may not receive enough work and frequent transfers resulting in excessive interaction may be required. If too much work is transferred, then the sender may be out of work soon, again resulting in frequent transfers.
- When is the work transfer performed? For example, in receiver initiated load balancing, work may be requested when the process has actually run out of work or when the receiver has too little work left and anticipates being out of work soon.

74) Explain the parallel Algorithm Models.

Answer: An algorithm model is typically a way of structuring a parallel algorithm by selecting a decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

### 1 The Data-Parallel Model


The data-parallel model is one of the simplest algorithm models. In this model, the tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data. This type of parallelism that is a result of identical operations being applied concurrently on different data items is called data parallelism. The work may be done in phases and the data operated upon in different phases may be different. Typically, data-parallel computation phases are interspersed with interactions to synchronize the tasks or to get fresh data to the tasks. Since all tasks perform similar computations, the decomposition of the problem into tasks is usually based on data partitioning because a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance.

Data-parallel algorithms can be implemented in both shared-address-space and message-passing paradigms. However, the partitioned address-space in a message-passing paradigm may allow better control of placement, and thus may offer a better handle on locality. On the other hand, shared-address space can ease the programming effort, especially if the distribution of data is different in different phases of the algorithm.

Interaction overheads in the data-parallel model can be minimized by choosing a locality preserving decomposition and, if applicable, by overlapping computation and interaction and by using optimized collective interaction routines. A key characteristic of data-parallel problems is that for most problems, the degree of data parallelism increases with the size of the problem, making it possible to use more processes to effectively solve larger problems.

An example of a data-parallel algorithm is dense matrix multiplication.

### 2 The Task Graph Model



In the task graph model, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs. This model is typically employed to solve problems in which the amount of data associated with the tasks is large relative to the amount of computation associated with them. Usually, tasks are mapped statically to help optimize the cost of data movement among tasks. Sometimes a decentralized dynamic mapping may be used, but even then, the mapping uses the information about the task-dependency graph structure and the interaction pattern of tasks to minimize interaction overhead. Work is more easily shared in paradigms with globally addressable space, but mechanisms are available to share work in disjoint address space.

Typical interaction-reducing techniques applicable to this model include reducing the volume and frequency of interaction by promoting locality while mapping the tasks based on the interaction pattern of tasks, and using asynchronous interaction methods to overlap the interaction with computation.

Examples of algorithms based on the task graph model include parallel quicksort, sparse matrix factorization, and many parallel algorithms derived via divide-and-conquer decomposition. This type of parallelism that is naturally expressed by independent tasks in a task-dependency graph is called task parallelism.

### 3 The Work Pool Model

The work pool or the task pool model is characterized by a dynamic mapping of tasks onto processes for load balancing in which any task may potentially be performed by any process. There is no desired premapping of tasks onto processes. The mapping may be centralized or decentralized. Pointers to the tasks may be stored in a physically shared list, priority queue, hash table, or tree, or they could be stored in a physically distributed data structure. The work may be statically available in the beginning, or could be dynamically generated; i.e., the processes may generate work and add it to the global (possibly distributed) work pool.

In the message-passing paradigm, the work pool model is typically used when the amount of data associated with tasks is relatively small compared to the computation associated with the tasks. As a result, tasks can be readily moved around without causing too much data interaction overhead. The granularity of the tasks can be adjusted to attain the desired level of tradeoff between load-imbalance and the overhead of accessing the work pool for adding and extracting tasks.

### 4 The Master-Slave Model

In the master-slave or the manager-worker model, one or more master processes generate work and allocate it to worker processes. The tasks may be allocated a priori if the manager can estimate the size of the tasks or if a random mapping can do an adequate job of load balancing. In another scenario, workers are assigned smaller pieces of work at different times. The latter scheme is preferred if it is time consuming for the master to generate work and hence it is not desirable to make all workers wait until the master has generated all work pieces. In some cases, work may need to be performed in phases, and work in each phase must finish before work in the next phases can be generated. In this case, the manager may cause all workers to synchronize after each phase. Usually, there is no desired premapping of work to processes, and any worker can do any job assigned to it. The manager-worker model can be generalized to the hierarchical or multi-level manager-worker model in which the top-level manager feeds large chunks of tasks to second-level managers, who further subdivide the tasks among their own workers and may perform part of the work themselves. This model is generally equally suitable to shared-address-

space or message-passing paradigms since the interaction is naturally two-way; i.e., the manager knows that it needs to give out work and workers know that they need to get work from the manager.

While using the master-slave model, care should be taken to ensure that the master does not become a bottleneck, which may happen if the tasks are too small (or the workers are relatively fast). The granularity of tasks should be chosen such that the cost of doing work dominates the cost of transferring work and the cost of synchronization. Asynchronous interaction may help overlap interaction and the computation associated with work generation by the master. It may also reduce waiting times if the nature of requests from workers is non-deterministic.

## 5 The Pipeline or Producer-Consumer Model

In the pipeline model, a stream of data is passed on through a succession of processes, each of which perform some task on it. This simultaneous execution of different programs on a data stream is called stream parallelism. With the exception of the process initiating the pipeline, the arrival of new data triggers the execution of a new task by a process in the pipeline. The processes could form such pipelines in the shape of linear or multidimensional arrays, trees, or general graphs with or without cycles. A pipeline is a chain of producers and consumers. Each process in the pipeline can be viewed as a consumer of a sequence of data items for the process preceding it in the pipeline and as a producer of data for the process following it in the pipeline. The pipeline does not need to be a linear chain; it can be a directed graph. The pipeline model usually involves a static mapping of tasks onto processes.

Load balancing is a function of task granularity. The larger the granularity, the longer it takes to fill up the pipeline, i.e. for the trigger produced by the first process in the chain to propagate to the last process, thereby keeping some of the processes waiting. However, too fine a granularity may increase interaction overheads because processes will need to interact to receive fresh data after smaller pieces of computation. The most common interaction reduction technique applicable to this model is overlapping interaction with computation.

An example of a two-dimensional pipeline is the parallel LU factorization algorithm.

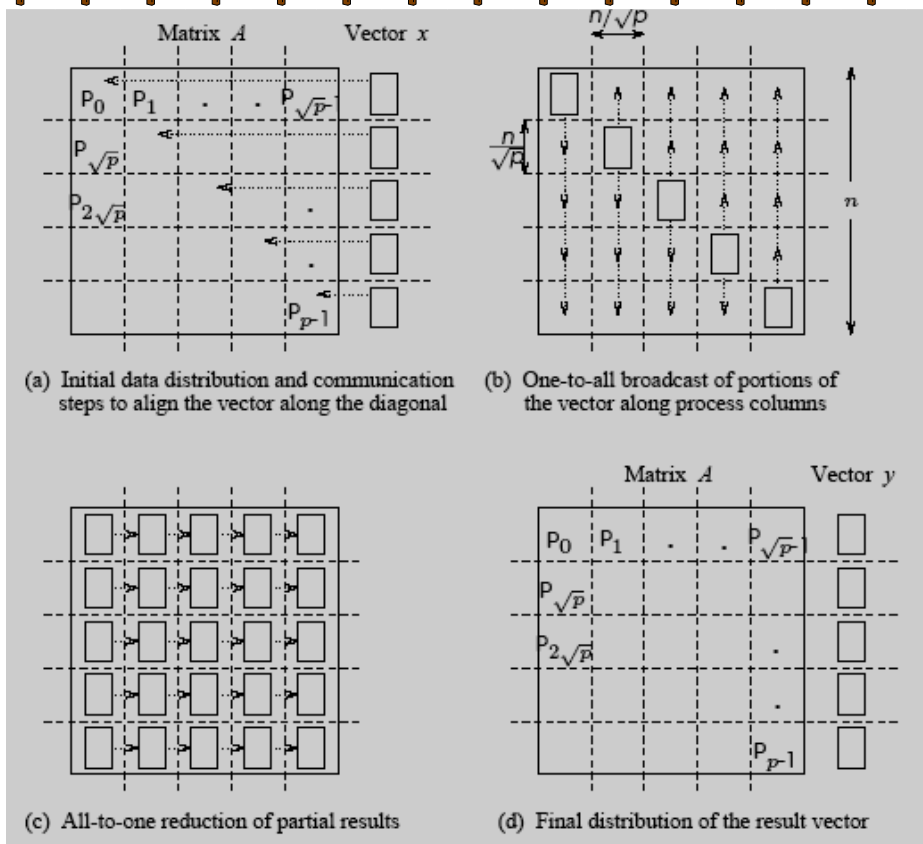
## 6 Hybrid Models

In some cases, more than one model may be applicable to the problem at hand, resulting in a hybrid algorithm model. A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm. In some cases, an algorithm formulation may have characteristics of more than one algorithm model. For instance, data may flow in a pipelined manner in a pattern guided by a task-dependency graph.

75) Describe a parallel formulation of Matrix-Vector multiplication using 2-D block partitioning.

Answer: **2-D Partitioning**

- The  $n \times n$  matrix is partitioned among  $n^2$  processors such that each processor owns a single element.
- The  $n \times 1$  vector  $x$  is distributed only in the last column of  $n$  processors.



Matrix-vector multiplication with block 2-D partitioning. For the

one-element-per-process case,  $p = \frac{n^2}{n}$  if the matrix size is  $n \times n$ .

- We must first align the vector with the matrix appropriately.
- The first communication step for the 2-D partitioning aligns the vector  $x$  along the principal diagonal of the matrix.
- The second step copies the vector elements from each diagonal process to all the processes in the corresponding column using  $n$  simultaneous broadcasts among all processors in the column.
- Finally, the result vector is computed by performing an all-to-one reduction along the columns.
  - Three basic communication operations are used in this algorithm: one-to-one communication to align the vector along the main diagonal, one-to-all broadcast of each vector element among the  $n$  processes of each column, and all-to-one reduction in each row.
  - Each of these operations takes  $\Theta(\log n)$  time and the parallel time is  $\Theta(\log n)$ .
- The cost (process-time product) is  $\Theta(n \log n)$ ; hence, the algorithm is not cost-optimal. When using fewer than  $n$  processors, each process owns an  $(n/\sqrt{p}) \times (n/\sqrt{p})$  block of the matrix.
  - The vector is distributed in portions of  $(n/\sqrt{p})$  elements in the last process-column only.
  - In this case, the message sizes for the alignment, broadcast, and reduction are all  $(n/\sqrt{p})$ .

The computation is a product of an  $(n/\sqrt{p}) \times (n/\sqrt{p})$  submatrix with a vector of length  $(n/\sqrt{p})$

- The first alignment step takes time
- $t_s + t_w n / \sqrt{p}$
- The broadcast and reductions take time

- $t_s + t_w n / \sqrt{p} (\log \sqrt{p})$
- Local matrix-vector products take time  $t_c n^2 / p$
- Total time is  $T_p \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$
- Scalability Analysis:
- $T_o = p t_p - W = t_s p \log p + t_w n \sqrt{p} (\log p)$
- Equating  $T_o$  with  $W$ , term by term, for isoefficiency, we have,  $W = K^2 t_w^2 p \log^2 p$  as the dominant term.
- The isoefficiency due to concurrency is  $O(p)$ .
- The overall isoefficiency is  $p \log^2 p$  (due to the network bandwidth).
- For cost optimality, we have,  $W = n^2 = p \log^2 p$ . For this, we have,  $p = O\left(\frac{n^2}{\log^2 n}\right)$

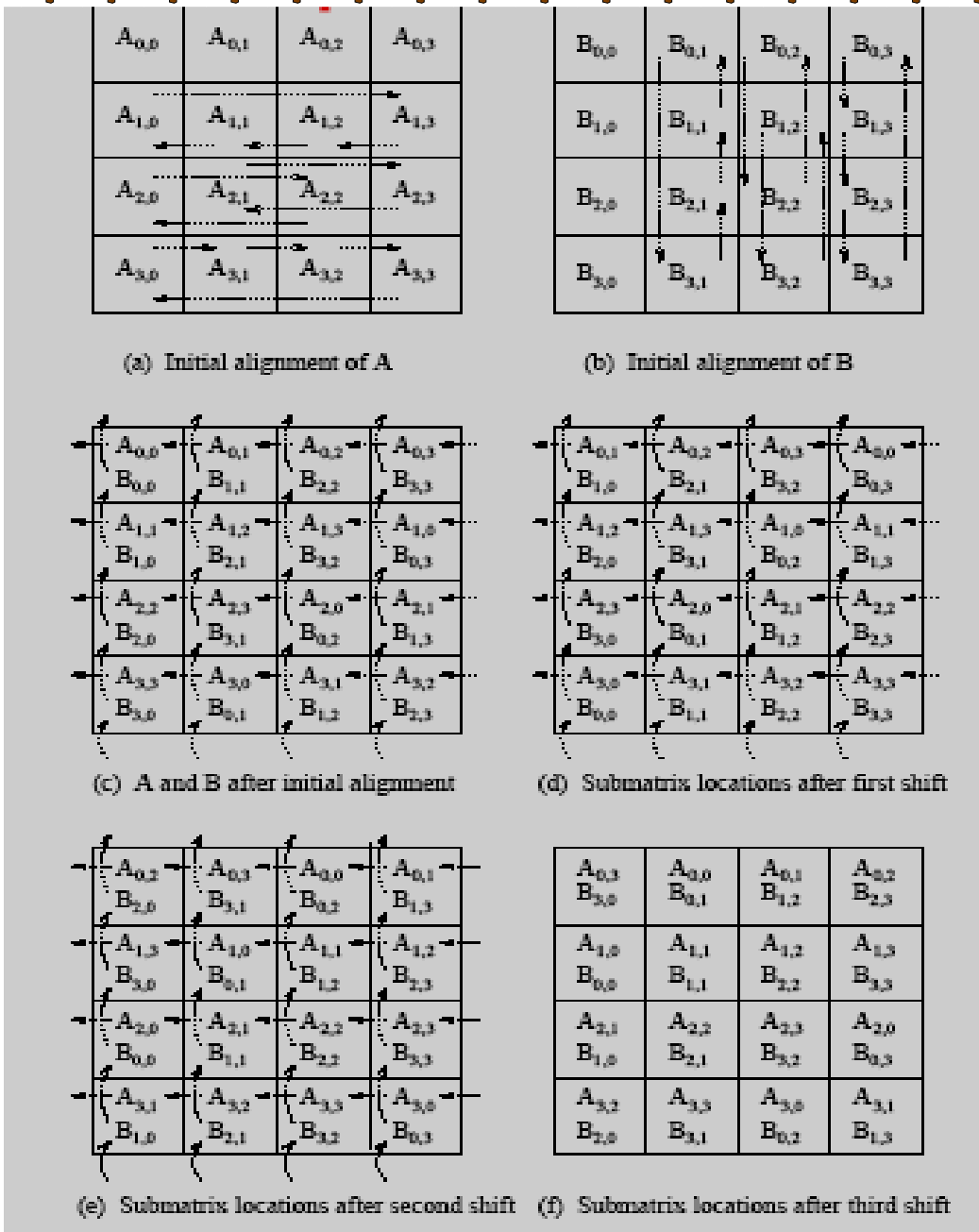
**76) Explain Canon's matrix-matrix multiplication.**

Answer: **Cannon's Algorithm**

- In this algorithm, we schedule the computations of the  $\sqrt{p}$  processes of the  $i$ th row such that, at any given time, each process is using a different block  $A_{i,k}$ .
- These blocks can be systematically rotated among the processes after every submatrix multiplication so that every process gets a fresh  $A_{i,k}$  after each rotation.

**Communication steps in Cannon's algorithm on 16 processes**

- Align the blocks of  $A$  and  $B$  in such a way that each process multiplies its local submatrices. This is done by shifting all submatrices  $A_{ij}$  to the left (with wraparound) by  $i$  steps and all submatrices  $B_{ij}$  up (with wraparound) by  $j$  steps.
- Perform local block multiplication.
- Each block of  $A$  moves one step left and each block of  $B$  moves one step up (again with wraparound).
- Perform next block multiplication, add to partial result, repeat until all  $\sqrt{p}$  blocks have been multiplied.
- In the alignment step, since the maximum distance over which a block shifts is  $\sqrt{p} - 1$ , the two shift operations require a total of  $2(t_s + t_w(n^2 / p))$  time.
- Each of the  $\sqrt{p}$  single-step shifts in the compute-and-shift phase of the algorithm takes  $t_s + t_w(n^2 / p)$  time.
- The computation time for multiplying  $\sqrt{p}$  matrices of size  $(n / \sqrt{p}) \times (n / \sqrt{p})$  is  $n^3 / p$
- The parallel time is approximately:  $T_p = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w \frac{n^2}{\sqrt{p}}$
- The cost-efficiency and isoefficiency of the algorithm are identical to the first algorithm, except, this is memory optimal.
-

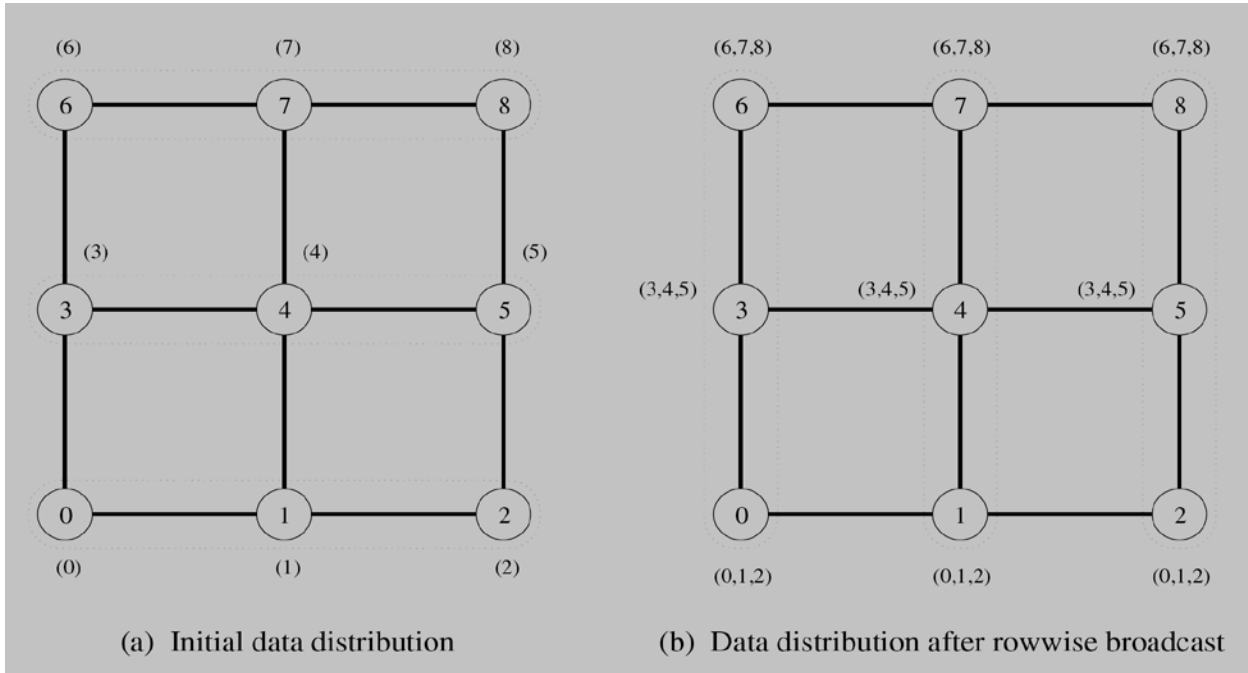


77) Explain all-to-all broadcast on a Mesh of p nodes.

**Answer:** Just like one-to-all broadcast, the all-to-all broadcast algorithm for the 2-D mesh is based on the linear array algorithm, treating rows and columns of the mesh as linear arrays. Once again, communication takes place in two phases. In the first phase, each row of the mesh performs an all-to-all broadcast using the procedure for the linear array. In this phase, all nodes collect  $\sqrt{p}$  messages corresponding to the  $\sqrt{p}$  nodes of their respective rows. Each node consolidates this information into a single message of size  $m\sqrt{p}$ , and proceeds to the second communication phase of the algorithm. The second communication phase is a columnwise all-to-all broadcast of the consolidated messages. By the end of this phase, each node obtains all p pieces of m-word data that originally resided on different

nodes. The distribution of data among the nodes of a 3 x 3 mesh at the beginning of the first and the second phases of the algorithm is shown in Figure .

Figure : All-to-all broadcast on a 3 x 3 mesh. The groups of nodes communicating with each other in each phase are enclosed by dotted boundaries. By the end of the second phase, all nodes get (0,1,2,3,4,5,6,7) (that is, a message from each node).



Algorithm 6 gives a procedure for all-to-all broadcast on a  $\sqrt{p} \times \sqrt{p}$  mesh.

**Algorithm 6 All-to-all broadcast on a square mesh of p nodes.**

```

1.  procedure ALL_TO_ALL_BC_MESH(my_id, my_msg, p, result)
2.  begin

/* Communication along rows */
3.      left := my_id - (my_id mod  $\sqrt{p}$ ) + (my_id - 1) mod  $\sqrt{p}$  ;
4.      right := my_id - (my_id mod  $\sqrt{p}$ ) + (my_id + 1) mod  $\sqrt{p}$  ;
5.      result := my_msg;
6.      msg := result;
7.      for i := 1 to  $\sqrt{p}$  - 1 do
8.          send msg to right;
9.          receive msg from left;
10.         result := result  $\cup$  msg;
11.     endfor;

/* Communication along columns */
12.     up := (my_id -  $\sqrt{p}$ ) mod p;
13.     down := (my_id +  $\sqrt{p}$ ) mod p;
14.     msg := result;

```



```

15.   for i := 1 to  $\sqrt{p} - 1$  do
16.       send msg to down;
17.       receive msg from up;
18.       result := result  $\cup$  msg;
19.   endfor;
20. end ALL_TO_ALL_BC_MESH

```

**78) Define and differentiate between ‘minimum execution time’ and minimum cost-optimal execution time.**

**Answer:** We are often interested in knowing how fast a problem can be solved, or what the minimum possible execution time of a parallel algorithm is, provided that the number of processing elements is not a constraint. As we increase the number of processing elements for a given problem size, either the parallel runtime continues to decrease and asymptotically approaches a minimum value, or it starts rising after attaining a minimum value. We can determine the minimum parallel runtime  $T_p^{\min}$  for a given  $W$  by differentiating the expression for  $T_p$  with respect to  $p$  and equating the derivative to zero (assuming that the function  $T_p(W, p)$  is differentiable with respect to  $p$ ). The number of processing elements for which  $T_p$  is minimum is determined by the following equation:

$$\frac{d}{dp} T_p = 0$$

Let  $p_0$  be the value of the number of processing elements that satisfies the above equation. The value of  $T_p^{\min}$  can be determined by substituting  $p_0$  for  $p$  in the expression for  $T_p$ . In the following example, we derive the expression for  $T_p^{\min}$  for the problem of adding  $n$  numbers.

**Example : Minimum execution time for adding  $n$  numbers**

The parallel run time for the problem of adding  $n$  numbers on  $p$  processing elements can be approximated by

$$T_p = \frac{n}{p} + 2 \log p$$

Equating the derivative with respect to  $p$  of the right-hand side of the equation to zero we get the solutions for  $p$  as follows:

$$-\frac{n}{p^2} + \frac{2}{p} = 0$$

$$\Rightarrow -n + 2p = 0$$

$$\Rightarrow p = \frac{n}{2}$$

Substituting  $p = n/2$  in the equation, we get

$$T_p^{\min} = 2 \log n$$

In the above example, the processor-time product for  $p = p_0$  is  $\Theta(n \log n)$ , which is higher than the  $\Theta(n)$  serial complexity of the problem. Hence, the parallel system is not cost-optimal for the value of  $p$  that yields minimum parallel runtime. We now derive an important result that gives a lower bound on parallel runtime if the problem is solved cost-optimally.

Let  $T_p^{\text{cost-opt}}$  be the minimum time in which a problem can be solved by a cost-optimal parallel system. If the isoefficiency function of a parallel system is  $\Theta(f(p))$ , then a problem of size  $W$  can be solved cost-optimally if and only if  $W = \Omega(f(p))$ . In other words, given a problem of size  $W$ , a cost-optimal solution requires that  $p = O(f^{-1}(W))$ . Since the parallel runtime is  $\Theta(W/p)$  for a cost-optimal parallel system, the lower bound on the parallel runtime for solving a problem of size  $W$  cost-optimally is

$$T_p^{\text{cost-opt}} = \Omega\left(\frac{W}{f^{-1}(W)}\right)$$

Example : Minimum cost-optimal execution time for adding  $n$  numbers

The isoefficiency function  $f(p)$  of this parallel system is  $\Theta(p \log p)$ . If  $W = n = f(p) = p \log p$ , then  $\log n = \log p + \log \log p$ . Ignoring the double logarithmic term,  $\log n \cong \log p$ . If  $n = f(p) = p \log p$ , then  $p = f^{-1}(n) = n/\log p \cong n/\log n$ . Hence,  $f^{-1}(W) = \Theta(n/\log n)$ . As a consequence of the relation between cost-optimality and the isoefficiency function, the maximum number of processing elements that can be used to solve this problem cost-optimally is  $\Theta(n/\log n)$ . Using  $p = n/\log n$  in the above equation, we get

$$T_p^{\text{cost-opt}} = \log n + \log\left(\frac{n}{\log n}\right) = 2 \log n - \log \log n$$

It is interesting to observe that both  $T_p^{\min}$  and  $T_p^{\text{cost-opt}}$  for adding  $n$  numbers are  $\Theta(\log n)$ .

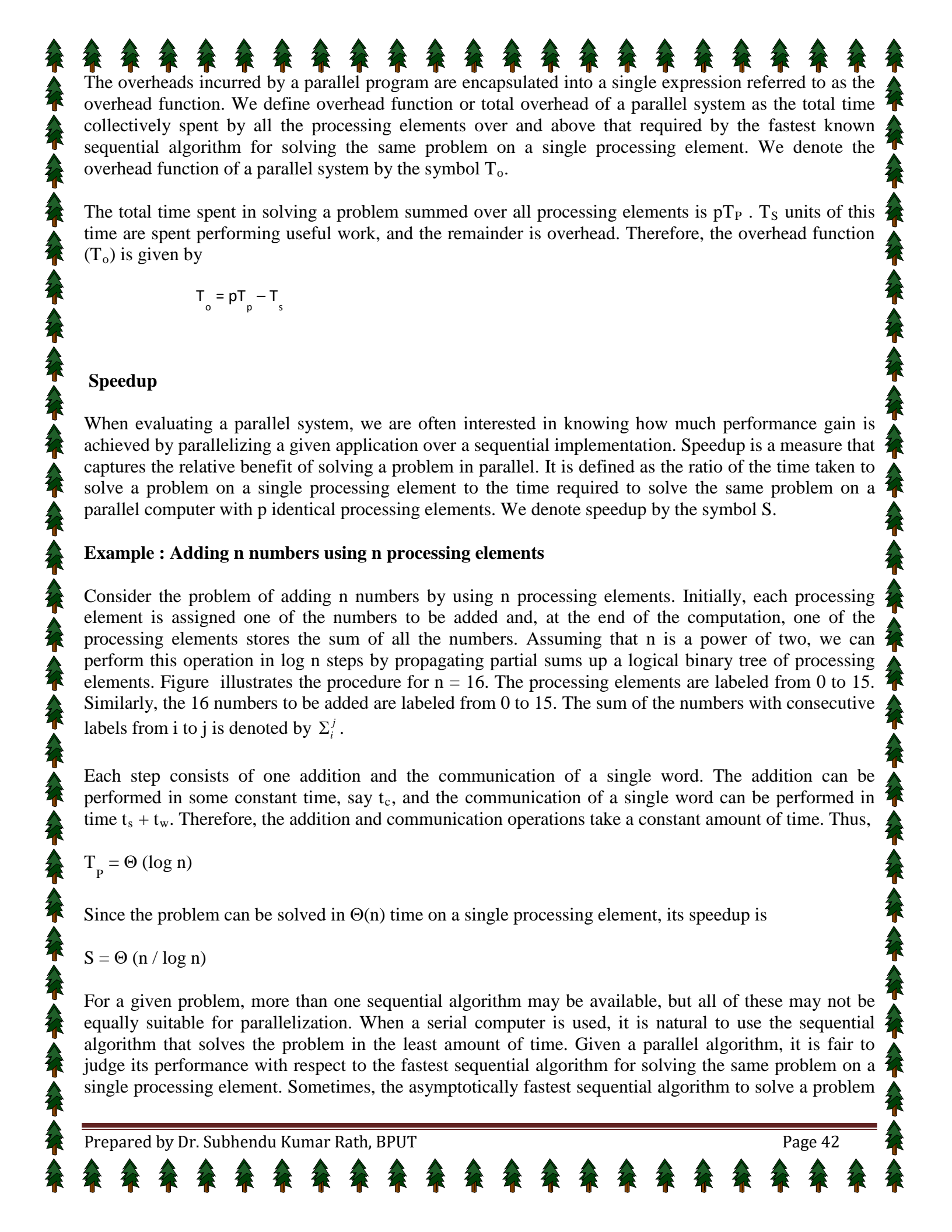
### 79) Explain the performance metric of parallel system.

**Answer:** It is important to study the performance of parallel programs with a view to determining the best algorithm, evaluating hardware platforms, and examining the benefits from parallelism. A number of metrics have been used based on the desired outcome of performance analysis.

#### Execution Time

The serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The parallel runtime is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes execution. We denote the serial runtime by  $T_S$  and the parallel runtime by  $T_P$ .

#### Total Parallel Overhead



The overheads incurred by a parallel program are encapsulated into a single expression referred to as the overhead function. We define overhead function or total overhead of a parallel system as the total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element. We denote the overhead function of a parallel system by the symbol  $T_o$ .

The total time spent in solving a problem summed over all processing elements is  $pT_p$ .  $T_s$  units of this time are spent performing useful work, and the remainder is overhead. Therefore, the overhead function ( $T_o$ ) is given by

$$T_o = pT_p - T_s$$

### Speedup

When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with  $p$  identical processing elements. We denote speedup by the symbol  $S$ .

### Example : Adding $n$ numbers using $n$ processing elements

Consider the problem of adding  $n$  numbers by using  $n$  processing elements. Initially, each processing element is assigned one of the numbers to be added and, at the end of the computation, one of the processing elements stores the sum of all the numbers. Assuming that  $n$  is a power of two, we can perform this operation in  $\log n$  steps by propagating partial sums up a logical binary tree of processing elements. Figure illustrates the procedure for  $n = 16$ . The processing elements are labeled from 0 to 15. Similarly, the 16 numbers to be added are labeled from 0 to 15. The sum of the numbers with consecutive labels from  $i$  to  $j$  is denoted by  $\Sigma_i^j$ .

Each step consists of one addition and the communication of a single word. The addition can be performed in some constant time, say  $t_c$ , and the communication of a single word can be performed in time  $t_s + t_w$ . Therefore, the addition and communication operations take a constant amount of time. Thus,

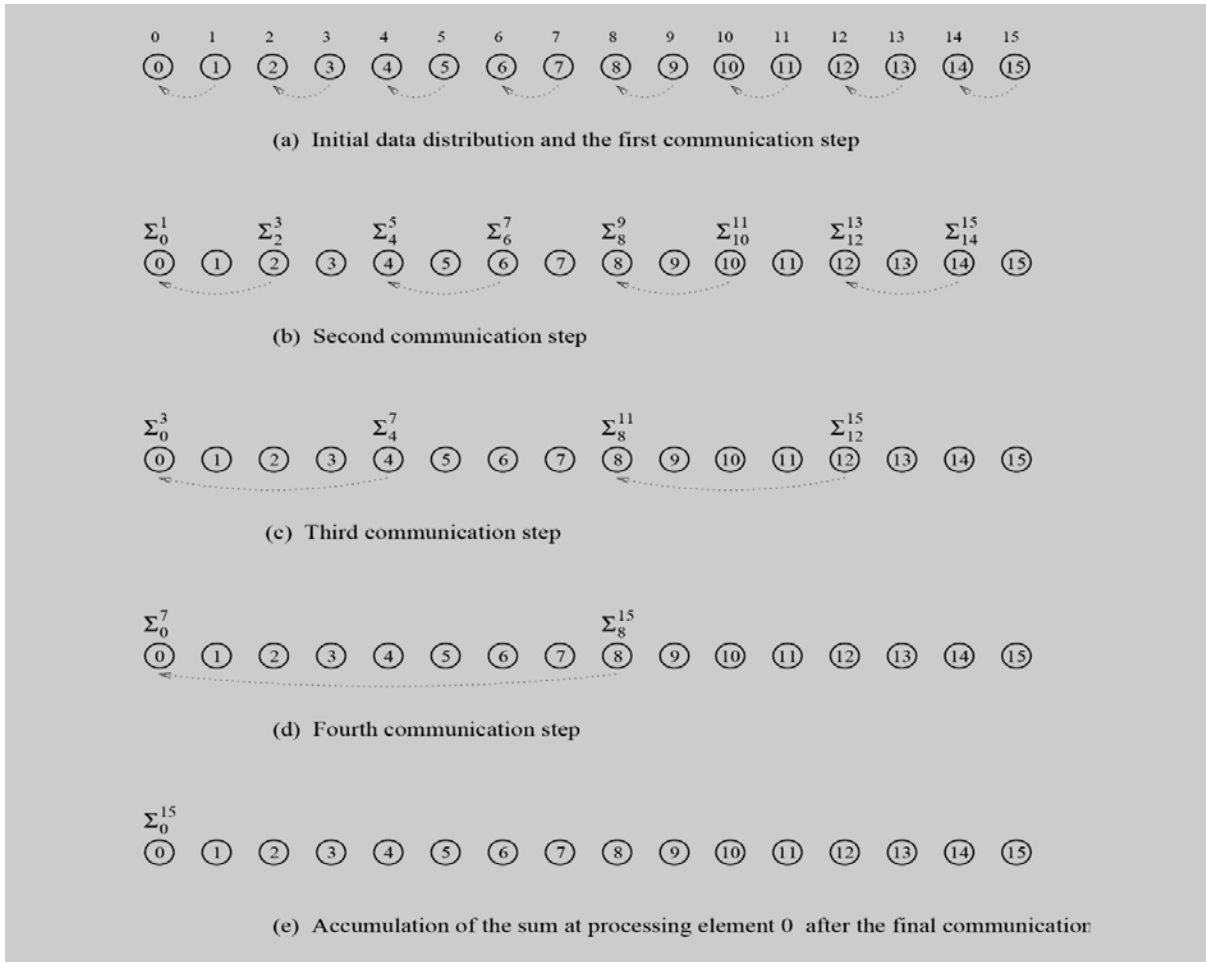
$$T_p = \Theta(\log n)$$

Since the problem can be solved in  $\Theta(n)$  time on a single processing element, its speedup is

$$S = \Theta(n / \log n)$$

For a given problem, more than one sequential algorithm may be available, but all of these may not be equally suitable for parallelization. When a serial computer is used, it is natural to use the sequential algorithm that solves the problem in the least amount of time. Given a parallel algorithm, it is fair to judge its performance with respect to the fastest sequential algorithm for solving the same problem on a single processing element. Sometimes, the asymptotically fastest sequential algorithm to solve a problem

is not known, or its runtime has a large constant that makes it impractical to implement. In such cases, we take the fastest known algorithm that would be a practical choice for a serial computer to be the best sequential algorithm. We compare the performance of a parallel algorithm to solve a problem with that of the best sequential algorithm to solve the same problem. We formally define the speedup  $S$  as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on  $p$  processing elements. The  $p$  processing elements used by the parallel algorithm are assumed to be identical to the one used by the sequential algorithm.



## Efficiency

Only an ideal parallel system containing  $p$  processing elements can deliver a speedup equal to  $p$ . In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm. As we saw, part of the time required by the processing elements to compute the sum of  $n$  numbers is spent idling (and communicating in real systems). Efficiency is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to  $p$  and efficiency is equal to one. In practice, speedup is less than  $p$  and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. We denote efficiency by the symbol  $E$ . Mathematically, it is given by


$$E = \frac{S}{p}$$

**Example : Efficiency of adding n numbers on n processing elements**

The efficiency of the algorithm for adding n numbers on n processing elements is

$$E = \Theta(n / \log n) / n = \Theta(1 / \log n)$$

We also illustrate the process of deriving the parallel runtime, speedup, and efficiency while preserving various constants associated with the parallel platform.

**Cost**

We define the cost of solving a problem on a parallel system as the product of parallel runtime and the number of processing elements used. Cost reflects the sum of the time that each processing element spends solving the problem. Efficiency can also be expressed as the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on p processing elements.

The cost of solving a problem on a single processing element is the execution time of the fastest known sequential algorithm. A parallel system is said to be cost-optimal if the cost of solving a problem on a parallel computer has the same asymptotic growth (in  $\Theta$  terms) as a function of the input size as the fastest-known sequential algorithm on a single processing element. Since efficiency is the ratio of sequential cost to parallel cost, a cost-optimal parallel system has an efficiency of  $\Theta(1)$ .

Cost is sometimes referred to as work or processor-time product, and a cost-optimal system is also known as a  $pT_P$ -optimal system.

**Example : Cost of adding n numbers on n processing elements**

The algorithm for adding n numbers on n processing elements has a processor-time product of  $\Theta(n \log n)$ . Since the serial runtime of this operation is  $\Theta(n)$ , the algorithm is not cost optimal.

----- THANK YOU -----