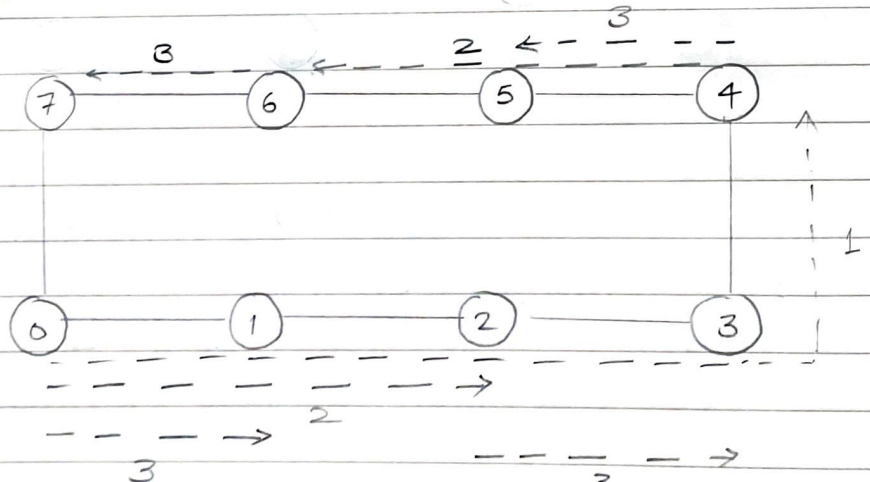
UNIT 3
PARALLEL COMMUNICATION.

* One-to-All Broadcast.

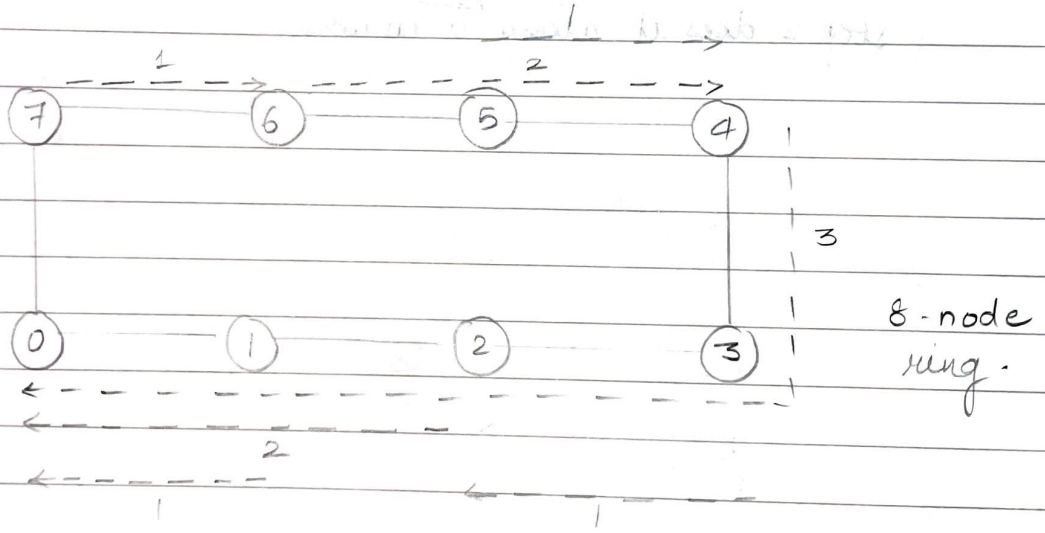
- Communication pattern where a single process or node sends a message to all other processes/nodes simultaneously.
- commonly used where multiple processors/nodes work together to solve a problem or perform a task
- sends a message to all other processors in the system.
- The message is replicated & delivered to each recipient process independently & concurrently.
- ensures efficient distribution of data cross the system, which enables efficient co-ordination & synchronization among the processes.



- Above is an example of one-to-all broadcast on an eight-node ring.
- Node 0 → source of the broadcast.
- message transfer step → dotted ~~line~~ arrow from source to destination
- no. on arrow → ⏱ time step during which message is transferred.

★ All-to-One Reduction.

- All-to-one reduction in parallel communication is a process where multiple entities such as processors/nodes share ~~their~~ data with a specific entity, which then combines these values to create a single result.
- helps in summarizing data from different sources into a final outcome
- process involves each entity calculating its own value & sharing it with the designated processor responsible for reduction
- this ~~identity~~ is responsible for aggregating the data to produce the desired result.



★ Mesh

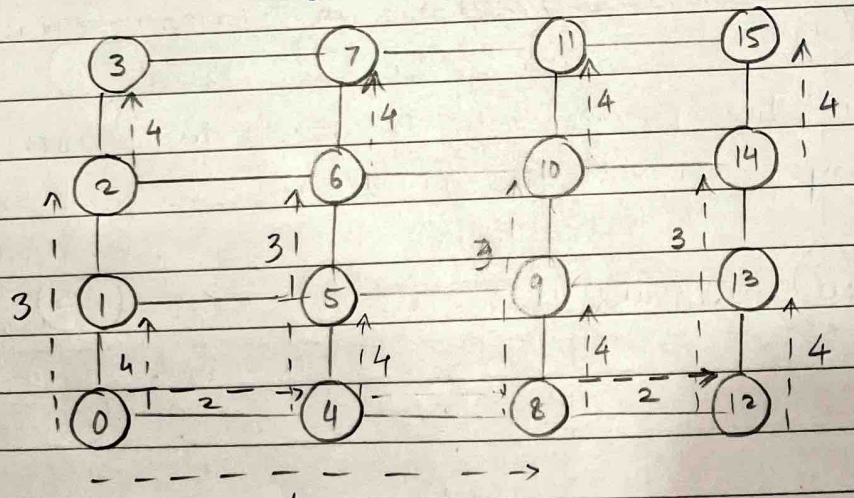
Broadcasting (One-to-all).

- To perform on a mesh network, broadcasting node starts by sending the message to ~~to~~ its neighboring nodes directly connected to it.
- each recieving node relays it to its own neighboring nodes.
- the process continues until data reaches all nodes in the network.

Algorithms used:

- Flooding
- Spanning Tree.

- **Flooding** - recieving node relays the message to all its neighbours, which results in a high degree of redundancy.
- Spanning tree - construct a tree structure over mesh network to minimize redundancy.
- each row & column of a square mesh ~~can be viewed of~~ p nodes can be viewed as a linear array of $\sqrt{p}$ nodes.
- Broadcast takes place in two steps:
  Step 1 does the operation along a row
  Step 2 does it along each a column.



One-to-all on 16-node mesh.

Reduction (All-to-one)

- To perform reduction, reduction operation (sum, prod, max, min) is applied to the data.
- reduction operation may involve exchanging values bet" neighboring nodes & progressively combining them until final result is obtained.

- Algorithm used → Tree based.
- nodes in the mesh are organized in a ~~mesh~~ tree structure reduction is performed bottom-up in the tree where child nodes sends values to parent nodes & the parent node combines the value according to reduction operation.
- process is continued until final result reaches root of the tree.

HYPERCUBE.

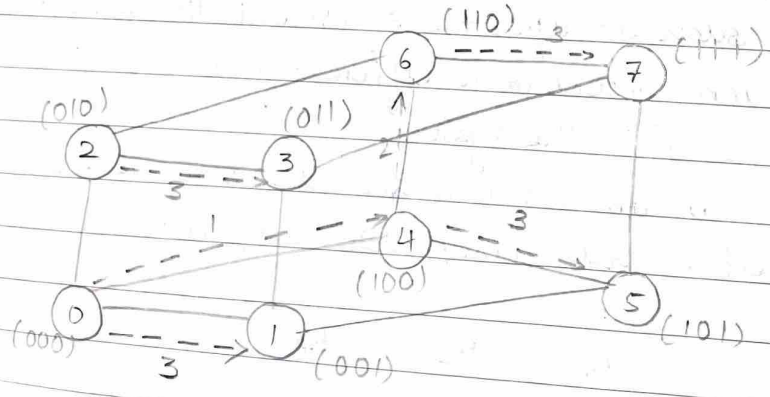
Hypercube is a type of network where - in each node is connected to exactly $\log_2 N$ other nodes,

N = total nodes in the network.

- Hypercube with 2^d nodes = d dimensional mesh w 2 nodes in each dim.

Broadcast

1) Source node initiates broadcast by sending messages to all of its neighboring nodes in the hypercube.
2) At each step, every node forwards the message to its neighbours that have higher dimension in common with the source node.
3) Process continues until all nodes recieve the message



$d = \log p$ steps.

Reduction

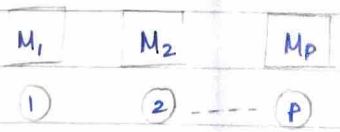
1) Each node starts with its own value / data

2) The nodes progressively reduce the no. of dimensions by pairing up with neighbors that share the same dimension.
3) In each step, the node with lower dimension sends its value to the node with the higher dimension.
4) The recieving node combines the recieved value with its own value using reduction operation.
5) This process continues until all values are sent to the root node, resulting in final reduced result.

ALL-TO-ALL BROADCAST & REDUCTION.

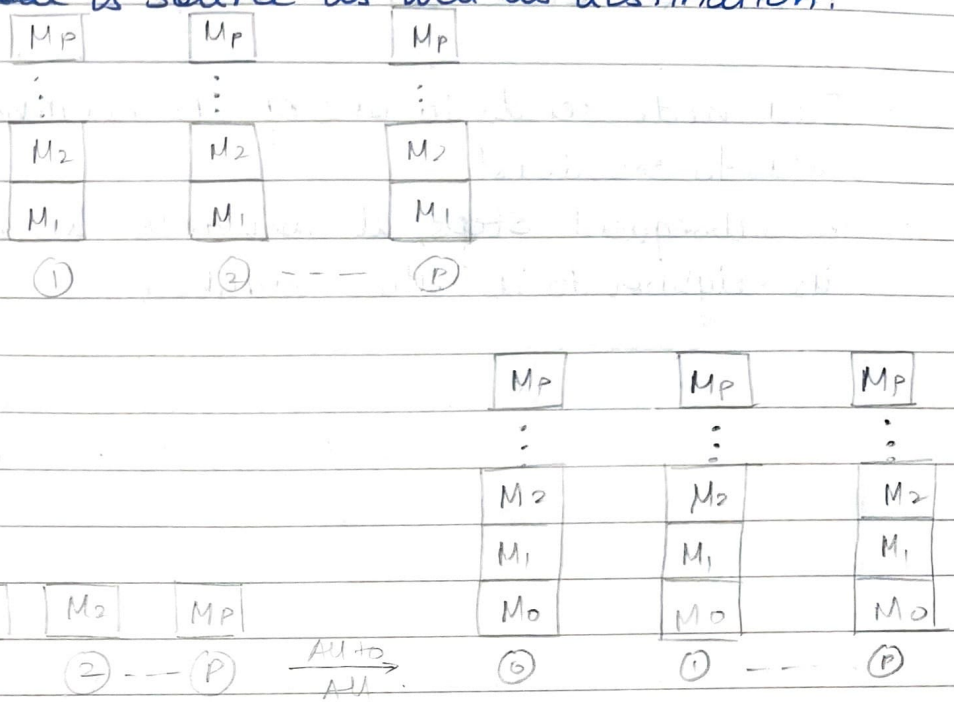
All-to-All Broadcast.

- generalization of broadcast in which each processor is the source as well as destination.
- message from a single source is sent to all other processes or nodes in the system.
- each process transmits its own message/data to all other processes simultaneously, resulting in every process recieving the same message.
- useful in scenarios where global information distribution is required such as updates, synchronization signals etc.
- allows each processor to efficiently communicate with all other processes in the system.
- ensures that ~~all~~ everyone has access to the same information.
- algo used → tree-based algo, ring algo etc.
- Eg.



Here, source node 1 sends message M1 to all other nodes also other p-1 nodes send their message (node 2-M2, node p, Mp) to other nodes including source node.

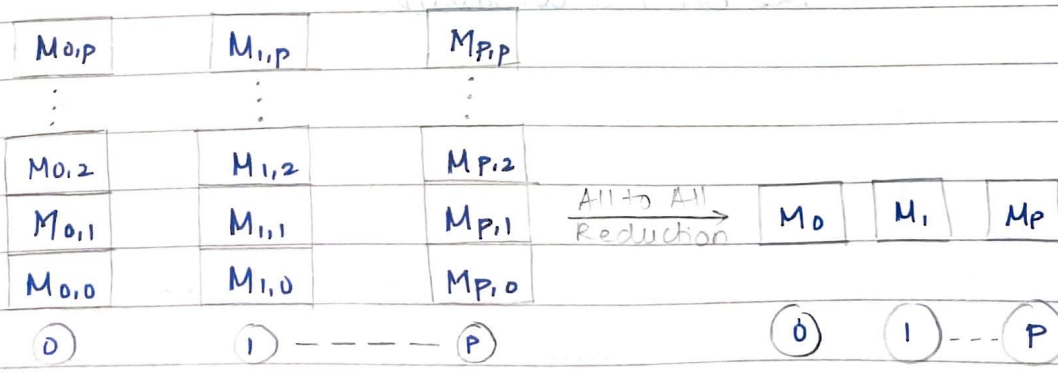
Here each node is source as well as destination.



All-to-All Reduction.

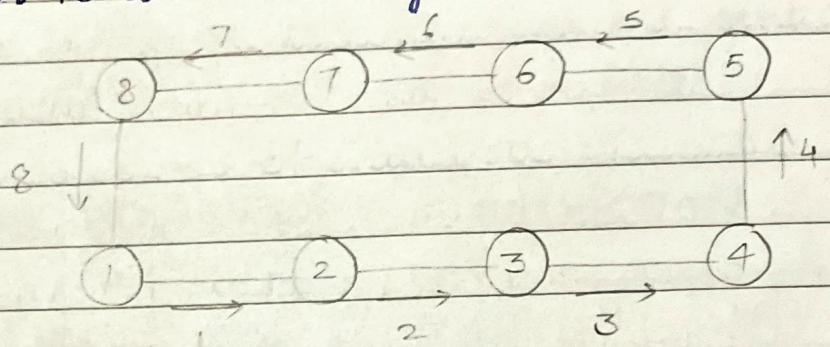
- reverse of all-to-all broadcast.
- every node contributes a value, reduction operation is performed collectively to obtain a result.
- the result is then distributed to other processes.
- result → single value that represents aggregation of all input values.
- Eg.

$$M_x = M_{0,x} \oplus M_{1,x} \oplus M_{2,x} \cdots \oplus M_{p,x}$$

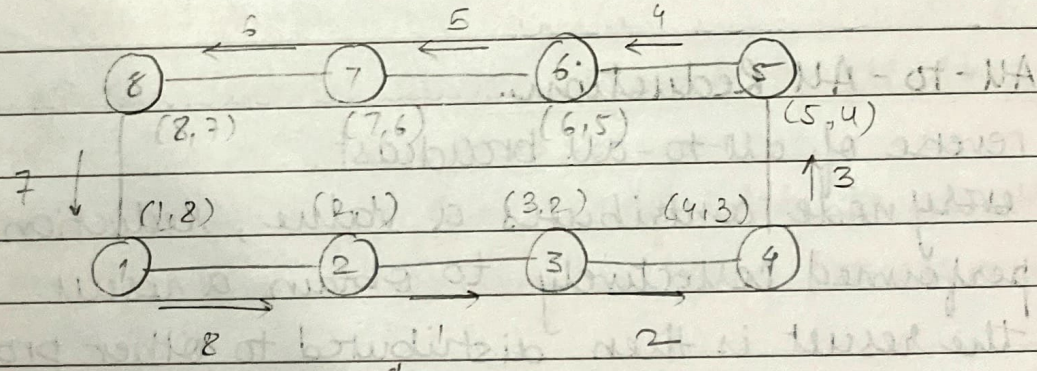


All-to-All Broadcast on Ring.

- Each node sends to one of its neighbors the data it needs to broadcast.
- In subsequent steps, it forwards the data recieved from its neighbor to its other neighbour.

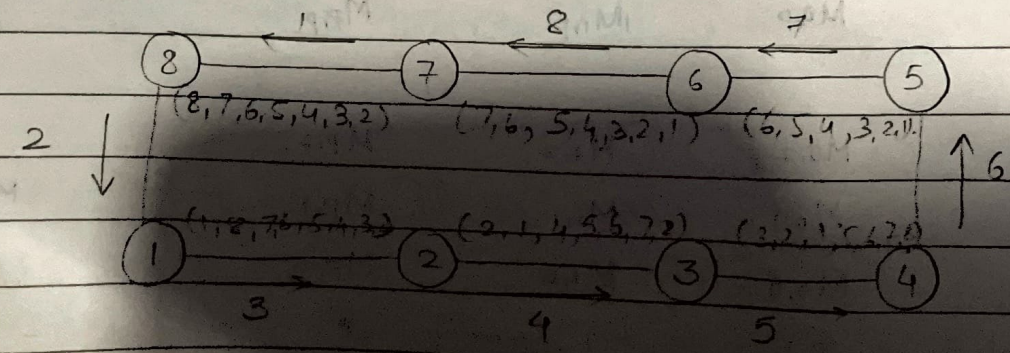


1st Communication step.



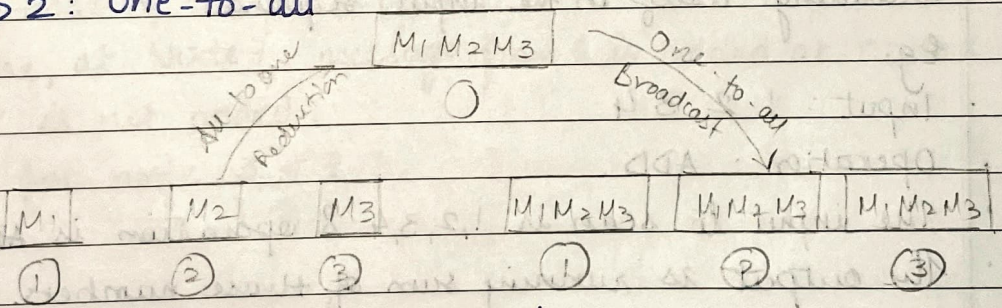
2nd Communication Step

Eg. node 8 has its own message & message it recieved from node 7 in earlier step.
Now this node broadcast these messages to its neighbor & the process continues.



All- Reduce.

- each node contributes a value & the values are combined using reduce operation.
- result is typically a single value that represents the aggregate of all input values.
- the result is then distributed back to all processes.
- Each process starts with its own input value.
- Define reduction operation to be performed on input values
- Execute the reduction operation in multiple rounds.
- In each round, processes exchange data with other processes & perform reduction operation
- Once reduction operation is complete, each process obtains final result.
- The result is then distributed back to all processes.
- Step 1: All-to-one reduction
  Step 2: One-to-all



- no node can finish reduction before each node has contributed a value.
- P processes : log p steps
  Total time : $T = (t_s + t_w m) \log p$.

Prefix-sum

- also known as scan.
- involves calculating cumulative sum of sequence of numbers

- transforms an input sequence to output sequence.
- each element in output sequence represents sum of all elements up to that position in the input sequence

Variants / Types.

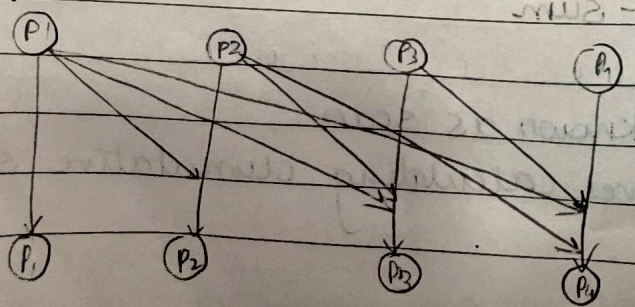
Exclusive | Inclusive.

1) Exclusive prefix-sum.
   - output sequence does not include the element at the current position.
   - each element represents the sum of all preceding elements in the input sequence
2) Inclusive prefix-sum
   - output sequence includes the element at the current position.
   - each element represents the sum of all preceding elements including itself in the input sequence

Eg.

- Input: 1 2 3 4
- Operation: ADD.
- The input to scan is 1,2,3,4 & operation is ADD.
- The output is running sum of those numbers.
- Output: 1 3 6 10
- Each output is sum of nos. from input up to that point.
- Prefix sum 6 is the sum of 1, 2 & 3
- Same prefix sum 3 is the sum of 1 & 2.

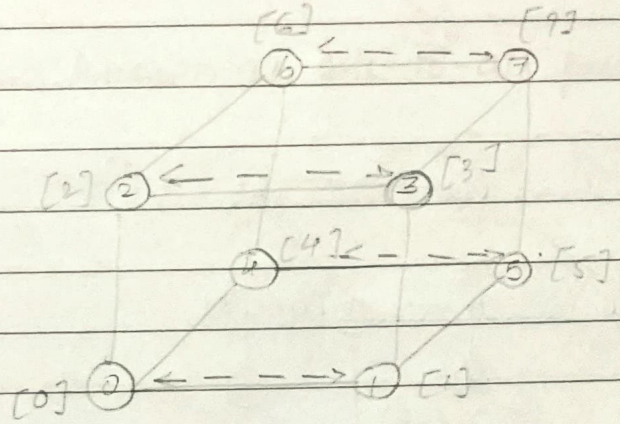
- Histogram uses prefix sum.
- Data compression.



* Example of Prefix Sum on Hypercube.

Step 1

The node with label K uses information from only k-node subset of those nodes whose labels are less than or equal to K.

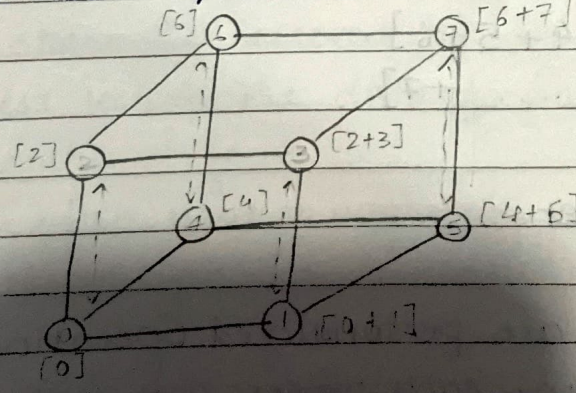


Step 2

Here content of incoming node is added to the result buffer only if message comes from a node with smaller label than that of recipient node.

So here, at Node 7, message from 6 is added at node 6, message of 1 is not added.

Same for node 3, 5 & 1.



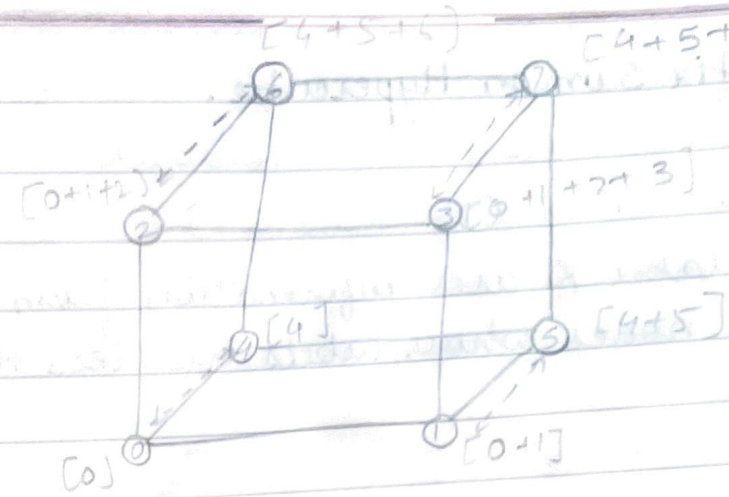
Step 3.

Here, result of buffer node

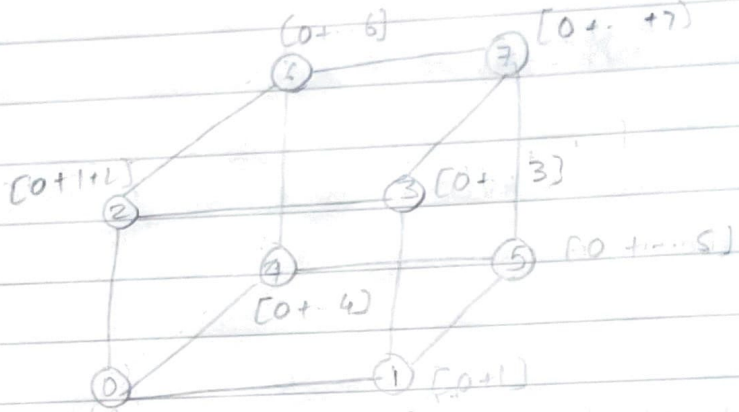
7 [4+5+6+7]

6 [4+5+6]

3 [0+1+2+3]



Step 4.



Result of all nodes:

N1 : [0+1]

N2 : [0+1+2]

N3: [0+1+2+3]

N4 : [0+1+2+3+4]

N5: [0+1+2+3+4+5]

N6: [0+1+2+3+4+5+6]

N7: [0+1+2+3+.... +7]

---

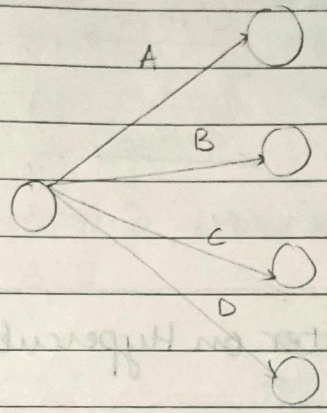
* Scatter & Gather.

- Scatter & gather are personalized operations
- Personalized means each process gets unique data.

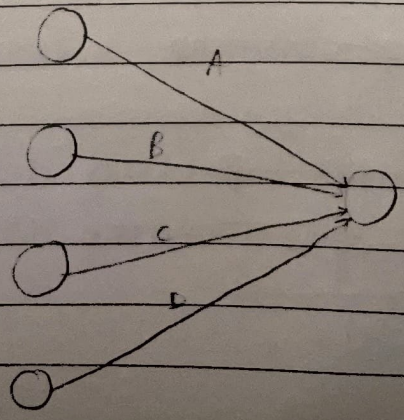
1) Scatter.

- Scatter is used to distribute data from a single process (the root source) to all other processes in the system
- The root process divides the data into equal-sized portions

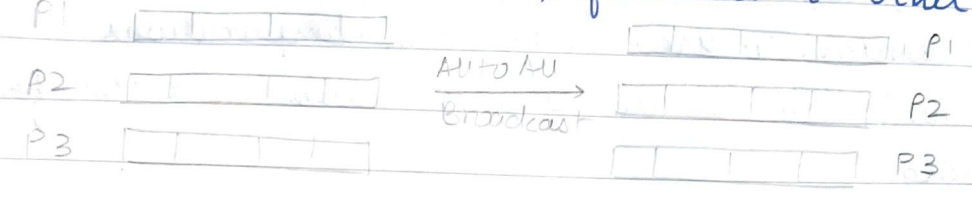
& sends each portion out to the corresponding process.
- Each recieving process recieves its portion of data which may be of different sizes depending on the distribution scheme.
- It is commonly used in parallel computations when the data needs to be divided among multiple processes for parallel processing.
- It is also known as one-to-all personalized communication.



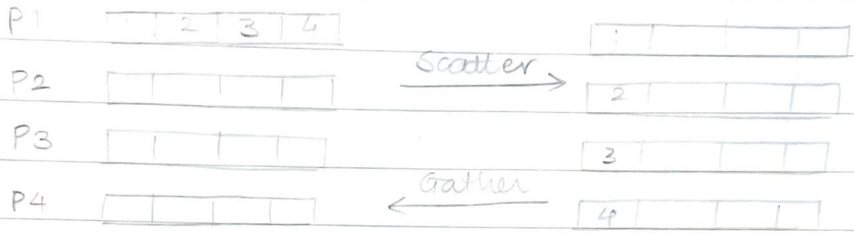
2) Gather.
- A single node collects unique value/data from each node.
- Gather is used to collect data from all processes in the system & brings it back to a single process.
- Each process sends its own data to the root process, which collects & stores the recieved data in a specific order.
- Root process recieves the data ~~for~~ from other processes & stores it in a combined form.
- Gather is often used to aggregate partial results for further processing or analysis.



- Scatter = broadcast (algorithmically)
  but scatter sends unique messages to other nodes whereas broadcast sends the same information to other nodes.



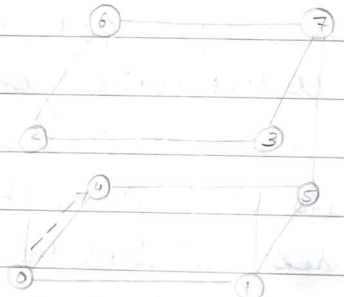
All-to-All Broadcast



Scatter & Gather

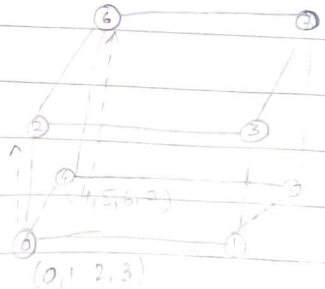
Example of Scatter on Hypercube

Step 1



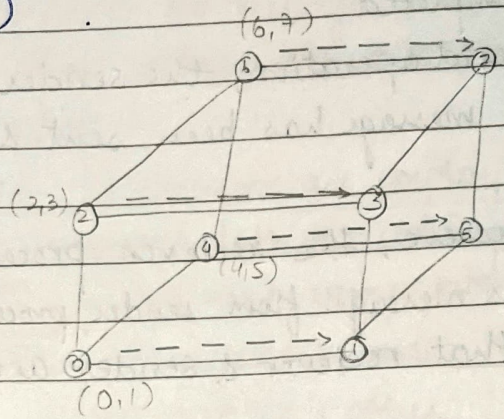
- In this step, the source transfers half of the data to one of its neighbors.
- Node 4 is the neighbor to node 0, so half of the data (4, 5, 6, 7) are transfered from Node 4 to Node 0.

Step 2



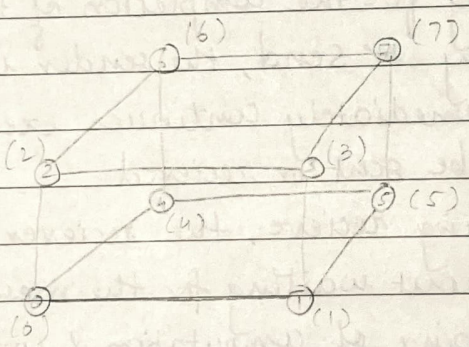
- In 2nd iteration, Node 0 & Node 4 transfers half messages to their neighbor Node 2 & 6.
- Node 2 : (2,3)
  Node 6 : (6,7)

Step 3



- In 3rd iteration, Node 0, 4, 2, 6 will transfer half data to their neighbors Node 1, 3, 5, 7.

Step 4.



Now all messages/data is scattered to every node.

Blocking & Non-Blocking MPI.

- MPI is Message Passing Interface.
- It is a popular communication library used in parallel computing to enable communication & co-ordination among processes.
- MPI provides both blocking & non-blocking communication modes to facilitate data exchange between processes.

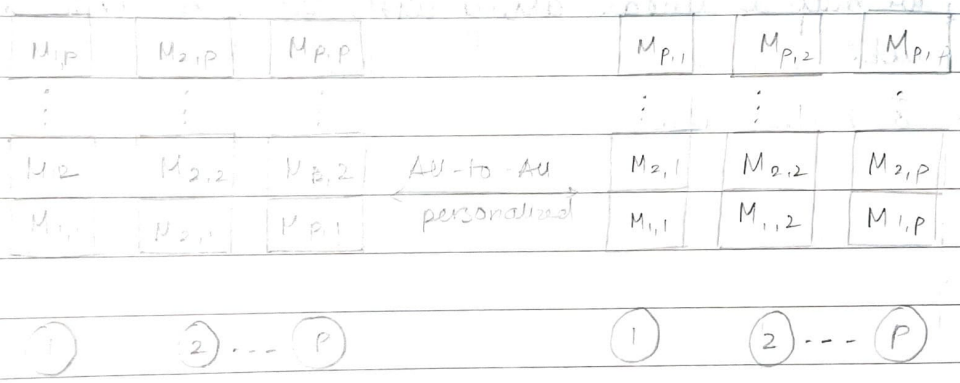
1) Blocking.
- default mode in MPI
- Sender & reciever processes are blocked until communication operation is completed.
- In blocking send operation, the sender process suspends execution until message has been sent & recieved by the reciever process.
- In blocking reciere, the reciever process suspends execution until it recieves message from sender process.
- makes sure that reciever & sender are ready for data exchange.

2) Non-blocking
- allows sender & reciever processes to continue execution without waiting for the completion of the communication.
- In non-blocking send, the sender initiates the send operation & immediately continues execution without waiting for message to be sent or recieved
- In non-blocking reciere, the reciever initiates reciere operation without waiting for the message to arrive.
- allows overlapping of computation & communication, potentially improving performance & reducing idle time.

All-to-all Personalized Communication.

- Also called Total Exchange
- each node exchanges a unique message with every other node in the system.
- Unlike traditional all-to-all ~~person~~ broadcast where same message is sent to all processes, all-to-all personalized communication allows each node to send distinct messages to individual recipients.

- Each process in the system starts with a set of input messages.
- Each node sends the unique data to corresponding recipient nodes
- Each node recieves the message/data sent by other nodes.
- The recieved data is then stored or processed as required by the application.
- Synchronization mechanisms are employed to ensure that all processes have completed both the sending & recieving phases before proceeding to the next computation..
- This ensures that all processes/nodes have exchanged their personalized messages with each other.



Circular shift.

- involves shifting the elements of a sequence/array in a circular shift manner.
- the elements are shifted to the right or left, (right)
- the last element wraps around to become the first element or
- the first element wraps around to become the last element. (left)
- used in matrix operations, string & image pattern matching.

i) Right Circular shift.

- In right circular shift, each element of the sequence is shifted one position to the right.
- The last element wraps around & becomes the first element

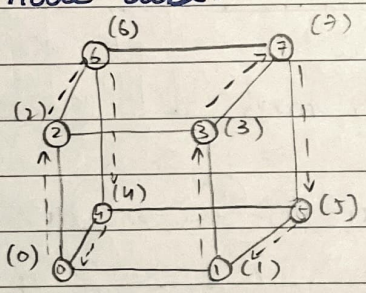
Eg. If we have a sequence [A, B, C, D, E] & perform a right circular shift, result → [E, A, B, C, D].

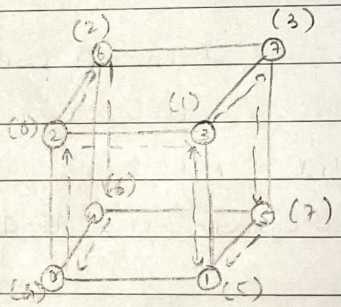
2) Left Circular shift.

- In a left circular shift, each element of the sequence is shifted one position to the left.
- First element wraps around & becomes the last element.
- Eg. [A, B, C, D, E] becomes [B, C, D, E, A].

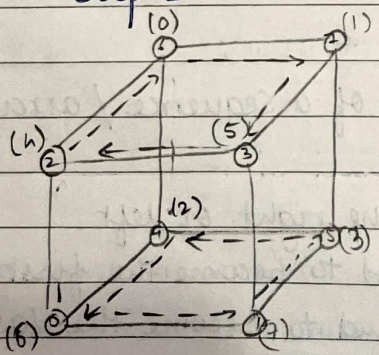
Circular shift on Hypercube.

- To develop a hypercube algorithm for the shift operation, we map a linear array with 2d nodes onto a d-dimensional cube.
- 8 node cube.

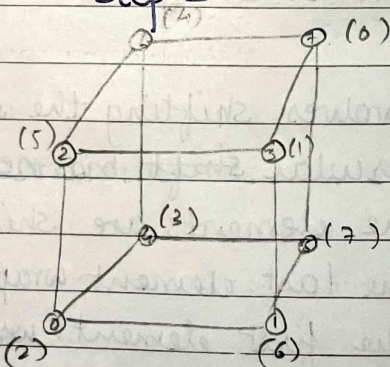




Step 1

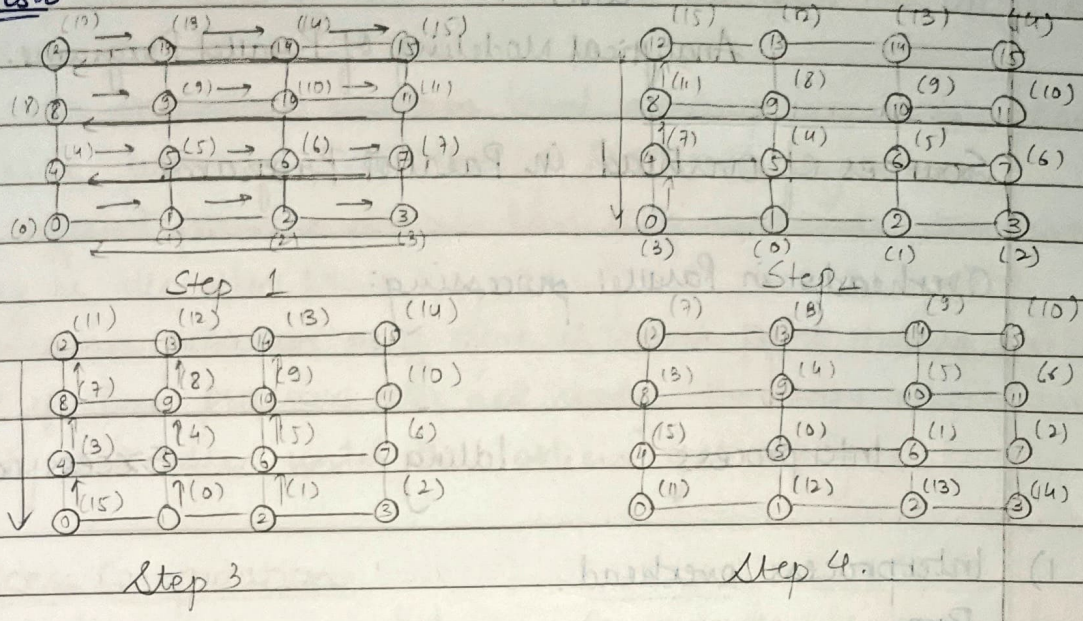


Step 2



- In each phase of communication, all data packets move closer to their respective destinations by short cutting the linear array in steps of power 2.
- All nodes can freely communicate in a circular fashion in their respective subarrays.

Mesh.



Improve the speed of communication operations.

1) One-to-all broadcast
- Splitting & routing messages into parts.
- Scatter + All-to-All broadcast
- To improve one-to-all, we first split the message into p parts (perform scatter operation), & follow it up by All to All broadcast.

  1 to all → Scatter → All to All.
- Scatter will divide the message into n parts, ~~every~~
- every node will carry each part of the message
- Now perform All-to-all broadcast.
- ∴ Every part of message will be broadcast from each node to all other nodes & we will get a complete message at each node.

2) All-to-one.
- All to one followed by gather operation.

  All-to-one + gather
- Messages are reduced to one message & then gather reduced ~~the~~ message.

UNIT . IV

Analytical Modeling of Parallel Programs.

Sources of Overhead in Parallel Programs.

Overheads in Parallel processing:

- Interprocess
- Idling
- Excess computation.

1) Interprocess overhead.
   - ~~Proc~~
   - Processors working on non-trivial parallel problems need to talk to each other.
   - Major source of parallel processing overhead is time spent to interact & communicate betⁿ processors.
   - If data dependency exists in problem, many processors (processors) work on the same data & therefore ~~they~~ use interprocess comm. like message passing.
   - This introduces communication overhead that impact the system
   - Using shared message IPC mechanism rather than message passing reduces overhead.

2) Idling:
   - State where one or more processing units are not actively executing any tasks.
   - this can happen due to many reasons such as load imbalance, synchronization etc
   - considered inefficient as all available processors are not being fully utilized.
   - In dynamic task generation app, it is very defficult

to predict size of the subtasks assigned to processors in advance.
- ∴ to maintain uniform load among processors, we cannot divide problem statistically bet" processing elements.
- If different processing elements have diff workloads, some processors may be idle when others are working
- Sometimes, processors must sync at some point during execution. But if some processors are not ready, the ones which are ready are idle until the others get ready.

3) Excess Computation
- It is the difference between computation performed by parallel system/program & computation for best serial algo.
- Excess = (comp. performed by ∥ program) - (computation for for best serial algo)
  Excess = Parallel - Best Serial.
- we try to generate a parallel program that works like a best serial program.
- Due to different factors parallel code takes more time than the best serial program & the net time is excess computation.

Performance Measures & Analysis.

1) Execution time.
- time needed to execute a task.
- denoted by Texe.
- aim is to add parallel hardware & reduce time.
- seqeuential/Serial time: time elapse bet^n start & end of execution by sequential computer (T_s)
- parallel: time elapse from the moment first processor starts to the last processor finishes execution.

2) Total Overhead.

- denoted by T_o.
- Time spent by parallel program - Time spent by fastest serial program.
- Overhead = $p(T_p - T_s)$
  p = no. of processors in parallel program.

3) Speedup

- how fast code runs in parallel as compared to serial execution
- evaluates benefit of solving a problem in parallel
- ratio of time taken ~~by a p~~ to solve on single processor to same problem on a parallel computer with p identical processors.
- $$S = \frac{T_s}{T_p}$$

  T_s = Time required on a single processor using fastest sequential algo.

  T_p = time reqd. using parallel processor.

$$\text{Speedup} = \frac{\text{Time for Serial}}{\text{Time for parallel}}$$

- can be as low as 0
- speedup greater than p is possible only if each processor takes less ~~tr~~ than T_s/p solving the problem.
- linear speedup → max speedup → p with p processors.

eg. Add^n of 16 numbers.

Time = T_s = 16 units

~~$T_s = O(n) = T_s t$~~

$T_s = \theta(n)$

$n = 16$

Time = 4 steps = 4 units = log16 = log n = 4
Speedup = $\Theta(n/\log n)$
= 16/4 = 4.

4) Efficiency.

- measure of fraction of time for which processing element is engaged.
- $$\text{Efficiency} = \frac{\text{Speedup}}{\text{No. of processors.}}$$
- Lowest efficiency = 0
- Highest efficiency = 1
- The speedup S of adding n numbers on n processors = $S = n/\log n$

$$\text{Efficiency} = \frac{\Theta(n/\log n)}{n}$$
$$= \Theta(1/\log n)$$

5) Cost.

- Cost of parallel algorithm is the product of running time & no. of processors.

Cost = Parallel running time × No. of processors.
$T_p \times p$.

- reflects the sum of time each processor takes to solve problem
- system is said to be cost optimal if cost of solving the problem on a parallel system is asymptotically identical to serial cost.
- Since $E = \frac{T_s}{p\,T_p}$

for cost optimal systems, $E = \Theta(1)$

Eg. Adding n nos. on n processors.

$T_p = \log n \quad (p = n)$.

Cost $\Rightarrow$ $pT_p = n \log n$.

Effect of Granularity on Performance.

- Granularity. - amt. of work between cross-processor dependencies.
- size or scale of the tasks that are divided & assigned to individual processors.
- ~~If the grain is larger~~
- larger grain is better & has fewer interactions; more local work can lead to load imbalance.
- if the grain is too fine, performance suffers from load imbalance.
- using few processors → improved performance of system.
- using fewer processors than maximum possible no. of processing elements is called downsizing.
- As no. of processors decrease by n/p, computation at each processor increases by n/p.

1) Load Balancing.
- Coarse grained: lead to load imbalance if workload not evenly distributed. → some units finish jobs faster & idling them.

Fine grained: allow more load balancing, reducing the chances of idle resources. imp. performance

2) Communication & Synchronization

Minimize
- Coarse grained: fewer sync & communication → less overhead
- fine grained: frequent communication & sync → high overhead due to increased interaction.

3) Parallelism - affects amt of II to be achieved.

Fine grained: more parallelism as smaller tasks can

be performed concurrently, but produce a lot of communication overhead which has a neg. effect on performance

Coarse Grained : limit the level of parallelism
~~increase have lower overhead but can still suffer from load imbalance~~ suitable for tasks that have inherent dependencies / cannot be divided.

4) Scalability.

- Fine-grained : allow fine-grained parallelism → scale well as the no of processors increase
increase communication overhead, limit scalability

Coarse-grained : lower overhead but suffer from load imbalance, limited parallelism which impacts scalability.

Scalability of Parallel Systems.

- Algos are scalable if the parallelism increases with increase in the size of the problem.
- architecture to implement an algo is scalable if it continues to yield the same output on each processor, even if the size of problem increases & no. of processors increase.
- data ~~scalable~~ parallel algo > architecture scalable algorithms
- To keep efficiency fixed, both size of problem & no. of processors must be increased.

Isoefficiency metric.

- Isoefficiency function : used to measure scalability.
- shows how the size of the ~~function~~ problem must grow as a function of no. of processors used to maintain constant efficiency.
- for a given problem size, as no. of processors ↑, efficiency of all processors ↓.

- for a system, if problem size ↑ & no. of processors is constant, efficiency ↓.
- function does not exist in unscalable parallel computing systems, as efficiency of these cannot be kept constant.
-

$$T_p = W + T_o(W, p)/p.$$

$$S = W/T_p$$

$$= \frac{W_p}{W + T_o(W, p)}$$

- The larger isoefficiency function → poor scalable system.

Minimum Cost & Minimum Execution time.

- If no. of processors is not a constraint, we can find out how fast a problem can be solved.
- ↑ in no. of processors result in ↓ execution time until it reaches an ~~asymptotic~~ minimum value or it starts rising after attaining a minimum value.
- Eq. to find no. of processors for which parallel runtime is minimum is

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

- Let p_0 = value of no. of processors.
- T_p^{min} can be found by substituting p_0 for p in exp. of T_p.
- Eg. Adding n nos.
  Parallel runtime is

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

Diff. w.r.t. p,

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

$$-n + 2p = 0.$$

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

Subs. $p = \frac{n}{2}$

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

Minimum Cost.

- refers to minimizing the overall execution time / resource usage which translates to improved performance & efficiency.
- let T_p^{min} → min time in which problem can be solved by cost optimal parallel system.
- isoefficiency of parallel is system is $\Theta(p \log p)$
- If,

$$W = n = f(p) = p \log p$$
$$\log n = \log p + \log p.$$

- If,

$$n = f(p) = p \log p$$
$$p = f(1-n)$$
$$= n/\log p \; n/\log n$$

Hence, $f^{-1}(W) = \Theta(n/\log n)$ ← max no. of processors used to solve.

- Put $p = \frac{n}{\log n}$ in $T_p = \Theta(\log n)$

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

$$= 2 \log n - \log \log n$$

- for adding 2 nos → cost opt. soln = asymptotically fastest soln.
- It is possible that $T_p^{\text{cost-opt}} > T_p^{min}$ for parallel systems.

Amdahl's Law.

- used for systems that have a fixed workload (comp).
- hence, as no. of processors increase, time required for execution must decrease.
- it quantifies that speedup can be achieved by parallelizing a computation.
- applies parallel processing to a task that contains both parallelizable & non-parallelizable portions.
- the law indicates that max. achievable speedup is limited by non-parallelizable portion of task.
- As no. of processors ↑, impact of non-parallelizable portion becomes more significant, reducing speedup.
- Defined by:

$$\text{Speedup} = \frac{1}{(1-P) + \left(\frac{P}{n}\right)}$$

P = computation which can be parallelized
n = no. of processors.

- formula assume that non-parallelizable portion remains constant regardless of no. of processing units used.
- Eg. consider a task having 80% of comp is parallelized ($P = 0.8$).

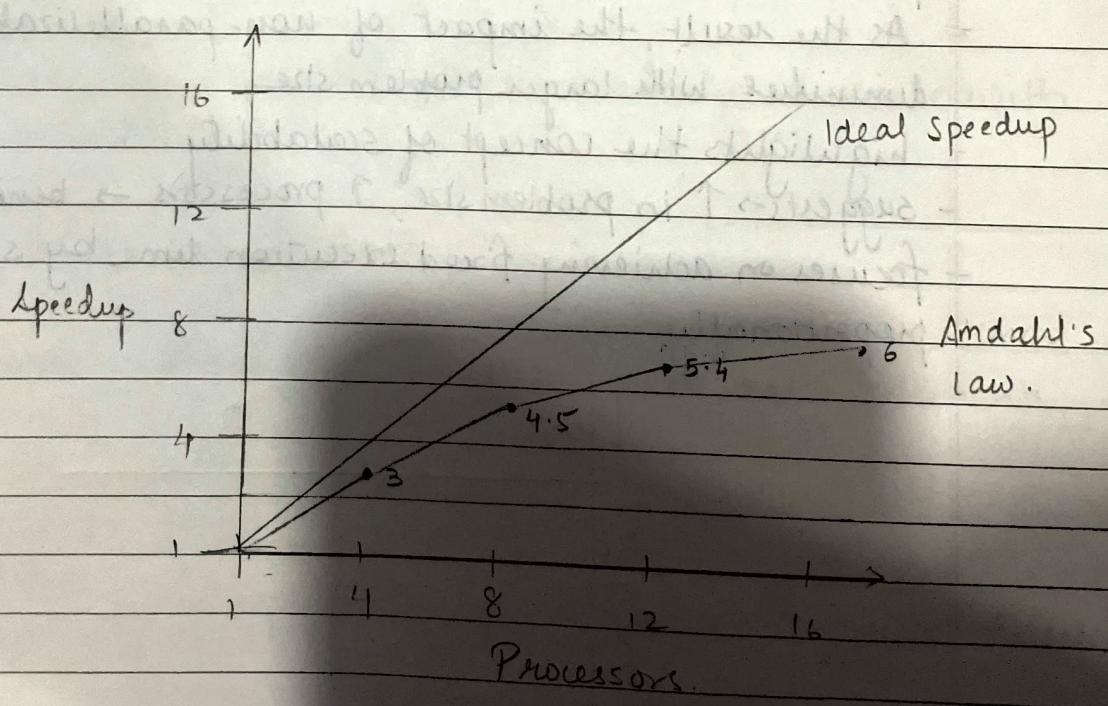
According to Amdahl's law, even with infinite processors, the max speedup is limited by non-parallelizable portion $(1-P)$, which is 20%. (0.2)

$$\text{Speedup} = \frac{1}{(1-0.8) + \left(\frac{0.8}{n}\right)}$$

as N approaches infinity, speedup converge to max of $\frac{1}{1-0.8} = 5$.

- highlights identifying & minimizing of non-parallelizable portion of task to max. speedup.

Cores	Speedup factor.			
1.	1.00 (Baseline)	sequential	potentially parallelizable	sequential
2.	$\frac{4 \text{ (seq)}}{3 \text{ (in this case)}} = 1.33$	sequential	core 1 / core 2	sequential
4.	$\frac{4 \text{ (seq)}}{2.5 \text{ (1.T.C)}} = 1.60$	sequential	core 1 / core 2 / core 3 / core 4	sequential
∞	$\frac{4 \text{ (seq)}}{2 \text{ (1.T.C)}} = 2$	sequential	sequential	



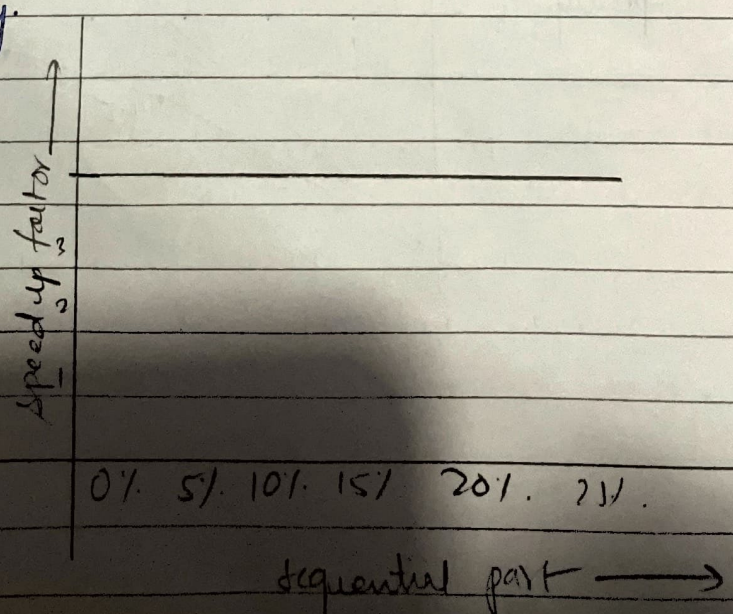
Gustafson's Law.

- overcomes drawback of Amdahl's law.
- relaxed the problem size from being fixed to be of any size.
- Instead of assuming we have fixed problem size, we must assume we have fixed execution time.
- because in case of increased problem size we need to increase no. of processors, not execution time.
- Given by

$$S(N) = N + (1-N) * P$$

S(N) = speedup as a func. of no. of processors (N)
P = proportion of computation which can be parallelized

- assumes that the non-parallelizable portion of the problem can be decreased with increase in size of the problem.
- As the result, the impact of non-parallelizable part diminishes with larger problem size.
- highlights the concept of scalability.
- suggests → ↑ in problem size, ↑ processors → benefits of PP leverage
- focuses on achieving fixed execution time by scaling problem proportionally.



Asymptotic Analysis of Parallel Programs.

- used to analyze efficiency & performance of algos as input size tends to infinity.
- provides insights into scalability & behaviour of program as no. of processors & problem size increases.
- Key considerations:

1) Time Complexity
   - analyze execution time of ~~processo~~ tasks & considering overhead.
   - A.A. gives insights into how exec. time scales with problem size & no. of units.
2) Scalability.
   - A.A. helps to identify whether program exhibits desirable scaling properties or there are bottlenecks that limit perf.
3) Work & Span.
   - technique used to evaluate efficiency
   - A.A of work & span can reveal inherent parallelism.
4) Parallel Overhead.
   - A.A. helps to asses ~~overhed~~ how overhead scale with no. of processors.

Dense Matrix Algorithms.

- dense matrix: ~~&~~ have few or almost ~~zero~~ no zero elements.

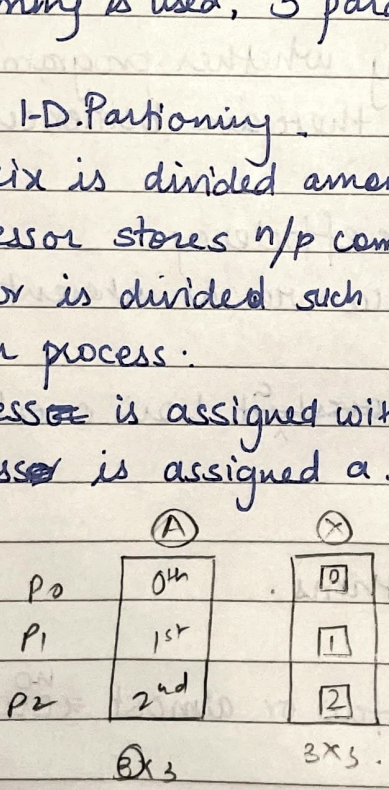
Matrix Vector Multiplication.

- involves multiplying matrix with a vector to produce resulting vector.
- matrix is 2D array of nos, vector is 1D array.
- dim. of both should be compatible for multiplication.

- the no. of columns in matrix = no. of elements in vector.
- resulting vector will have same no. of rows as matrix.
- involves taking dot product of row with vector.

- We are going to multiply $n \times n$ matrix A with $1 \times n$ vector x to yeild $n \times 1$ result vector y.
- $y = y + A * x$. A = dense matrix.
- ~~algo~~ serial algo requires n^2 mult & add^{n}s.
- $\therefore$ sequential is $W = n^2$.
  runtime
- Depending upon whether row ~~is~~ 1D ~~to~~ ~~or~~, column wise 1D or 2D partitioning is used, 3 parallel formulations are possible.

* Row-wise 1-D Partioning.
- $n \times n$ matrix is divided among p processors, ~~each pro~~
- each processor stores n/p complete rows of matrix.
- $n \times 1$ vector is divided such that each processes has n/p elem.
- One row per process.
- Each processor is assigned with one row of the dense matrix A
- Each processor is assigned a single element of the vector x.



first row is assigned to process P_0 & 0^{th} element is given to P_0. second row → P_1, 1^{st} element P_1.
Similarly all other vectors & rows are assigned.