NO:

DATE: 17/02/24 Decomposition Techniques.

Gamble, Risk

4) Speculative Decomposition :-

⇒ Branch prediction.

5) Hybrid Decomposition :-

↳ we combine decomposition technique. (data, rewrrsive, ...)

Mapping Schemes :-

⇒ # of tasks
⇒ size of each task } → apriori (to know something before time)

⇒ if known, then Static Mapping → decides which task at which process

⇒ unknown, Dynamic Mapping → decides during execution.

Static causes load imbalance, know about size or # at time of actual execution.

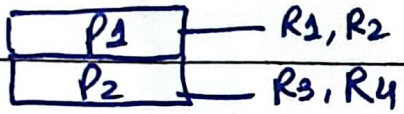
Static Mapping

NO:
DATE:

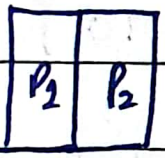
Array Distribution Schemes :-

Block distribution → 1D → (divide rows or columns)

row-wise dist



column-wise → n/p rows as block
→ n/p column



2D :-

$P = p1 * p2$ 4x4 grid 2x8

chunk size = $n/p1 * n/p2$

Cyclic :-

↳ no of chunks > # of p's

4x4 grid p=3 make sure no load imbalance

0 1 2

0 1 2

Block-Cyclic dist :- (issue)

NO:
DATE:

Sparse Matrix → Task Interact

issue → some P's idle, some have more load.

Sol:- randomize the mapping

V = {0 1 2 3 .... 11} elements.

random(V) = { 8, 2, 6, ..., 10 }

mapping = 8 2 6 ... ...
P0

Threads:-

↳ inter-process comm → more expensive

↳ inter-thread comm → less "

Pthread → POSIX

independent stream of ins.

⇒ Strcpy → doesn't do bound checks, so not a recommended fun → buffer overflow.

⇒ integer overflow:-

OpenMP:-

NO:

DATE: 19th Feb 25

override default value

omp_set_num_thread(4); ←

OpenMP: → by default [4] threads.

↳ we can set environment variable e.g it can be 2, 4

omp_get_thread_num()

↳ return thread id of threads.

⇒ must include <omp.h>

In visual Studio, enable checkbox OpenMP.

```
#pragma omp parallel
{
    // run in parallel
}
```

output: hello(0) word(0)
hello(1) world(1)
hello(2) world(2)
hello(3) world(3)

#pragma omp parallel num_threads(4) ....

directives.

get # of threads actually created ← open_get_num_threads()

NO:

DATE:

Loop work sharing construct:-

Sequential code add 1D array

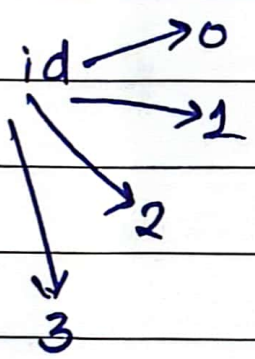
open mp parallel region

master thread

↓

first thread always has ID value = 0

→ give array Parts to threads.



istart = 0 iend = 25

istart = 25 iend = 50

istart = 50 iend = 75

istart = 75 iend = 100

In-built construct for FOR loop of open mp

```
# pragma omp parallel
  # pragma omp for
  for(i=0; i<N; i++){
    a[i] = a[i] + b[i];}
```

can write here no need for separate line

→ omp assign itself to every threads in proper way.

NO:
DATE:

Fork-join model :-

Compile:

Linux:

g++ -fopenmp hello-world.cpp

-fopenmp : open mp enabled.

Pi :-

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

⇒ break the curve into very small curves or rectangles and cal area of □s.
we can do by openmp.

$$\sum_{i=0}^{N} F(x_i)\,\Delta x \approx \pi$$

$F(x_i)$ → height; Δx → width; x_i → middle of each interval

Step = the width of each □

$$= \frac{1}{100,000}$$

$i=0 \rightarrow x = \frac{0.5}{100,000}$

$i = 1 \quad x = \frac{1.5}{100,000}$

$i=2 \quad x = \frac{2.5}{100,000}$

NO:
DATE:

⇒ nthreads = omp_get_num_threads (good practice use it instead of just using #define Num_threads 4.

Synchronization :-

mutual exclusion :- only one thread at a time can enter critical section.

```
# pragma omp critical [name]
{
    code-block
}
```

- [name] optional name that identifies critical directive.

Previously sum array. but now sum in critical region so no issues due to sum → only one { can update (Pi) at a time

also its initialized inside parallel so they are private to each thread.

↑ separate copy of every thread.

NO:
DATE:

OpenMp → env variable
→ directives
→ library functions

→ openMp for constructs.
→ openMp parallel section
→ openMP critical section.

Static Scheduling

pragma omp for schedule (static, chunk-size)

if not specified, it will split into equal chunks e.g
i=100
chunk-size=25

e.g
i = 100
chunk size = 10
means 10 iterations in one thread.
(total e.g 4 threads)

10 20 30 40
T1 T2 T3 T4
50 60 70 80
90 100

NO:

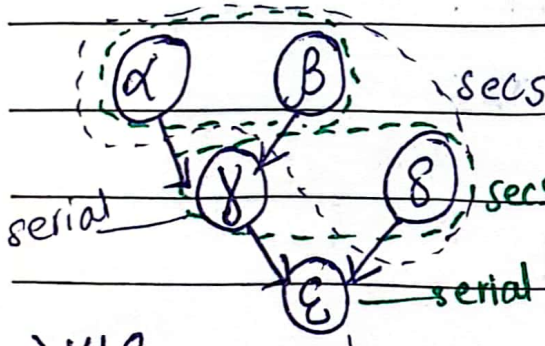
DATE:

⇒ if last private not set, then S value is value that set before #pragma.

⇒ Conditional parallelism :-

if (<scalarexpression>)

⇒ OPEN MP task construct :-



⇒ we can use sections. execute pragma #omp single

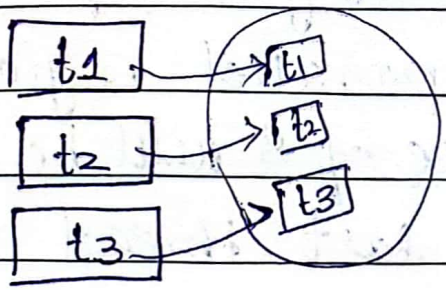
only one thread will do it

⇒ linked-list traversal

↓

cannot do by sections

so OMP → tasks



→ tasks forming by while loop → made by one thread.

→ put in (addon) pool → their execution by dynamically idle threads.

NO:
DATE:

firstprivate

e.g firstvariable(s)

master thread set value will be assigned to s (private variable) of threads initially.

```
# pragma omp parallel section
    section → th0
    section → th1
# pragma omp parallel
# pragma omp sections
    section → th0
    section → th1
```

lastPrivate:

e.g lastprivate(s)

→ out of parallel region then value of s will be last update. (e.g last thread done work on s)

NO:

DATE:

Private

pragma omp parallel for private(j)

⇒ i will be shared

every thread will have own copy of j

har thread ka private ban jayega

before ↓ (shared) var, parallel then by default

after, then by default (private)

by default (private) ← iteration kay liye use hota ha

default(none)

↓

inforce programmers to tell about each var either its private or shared, good prog practice.

*const var, don't write priv or shared

⇒ default(shared)

↳ if not mention, then by default shared

NO:

DATE: 5th March '25

Scheduling loops

1) Static:

[ ] (optional)

Static, 10 → 10 iterations thread.

schedule (static [, chunk-size])

if not mention, it divides itself equally to # of threads.

2) Dynamic:

↓ at run time

if NO chunk size, then default size = 1

Schedule (dynamic [, chunk-size])

3) Guided: It decides chunk size heuristically.

schedule (guided)

if no chunk size = then min 1

schedule (guided , c)

↳ chunk size

schedule (runtime) OMP_SCHEDULE

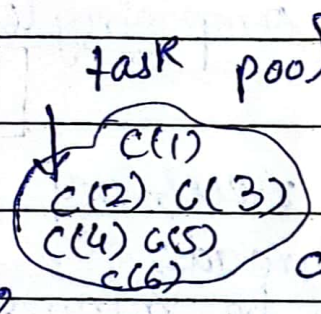
↓ value of variable either static or dynamic

→ environment variable

NO:
DATE:

```
omp_set_num_threads(2);
#pragma omp parallel default(none)
{
    #pragma omp single
    {
        #pragma omp task
        fprintf(stderr, "A\n");
        #pragma omp task
        fprintf(stderr, "B\n");
    }
}
```

into pool, execution separate from creation of tasks.



thread 0: a, C(1), C(7), C(5), z

thread 1: C(2)

thread 2: C(3) C(6)

thread 3: C(4)

z ↓ out of parallel region so executed by master thread.

Distributed Systems

NO:

DATE:

12 March 2025

based on map-reduce model → e.g Hadoop → big data
Apache Spark

Google PageRank algo

IOT (Internet of things)

• Cloud computing

Cluster Computing :-

↳ set of comps, near to eachothers, connected, performing like they are single machine.
fast wires e.g 10 Gbps

Grid computing :

→ interconnected but owned by multiple organizations

↓ mostly for scientific applications

On-prem
"on-premises"

cloud computing ← switch to

due to means once gone can't be recovered

(i) cost is less (a lot cheaper)
(ii) sunk cost (initially spend to start business in On premises)

NO:

DATE:

Monte Carlo Method :-

↳ randomized algo

↓ generate many points in square

↓ in circle or out of circle (inside square must)

$$\Rightarrow \boxed{(x-a)^2 + (y-a)^2 < r^2}$$

(a = 0.5, r = 0.5, r^2 = 0.25)

① calcute # of points in circle

② divide points in ○ and × 4 ③

Point in □ → total points actually.

as # of random value increase, $\boxed{\pi}$ value becomes more accurate.

parallize → divide iterations.

thread ⓪

```
# pragma omp master
```

↓ master thread will execute it (like single)

reasonable # of threads on upper layer are enough.

NO:

DATE:

Nested parallelism → TRUE
→ not good choice

OMP_NESTED → False (default)
→ True

```
# pragma omp parallel num-threads(4)
{ ...
    # pragma omp parallel num threads(2)
    }
```

to t1 t2 t3

→ this decides if new team of threads created or not or original used.

if FALSE

to → [to,0 to,1]

not creating new threads, image of multithreading but not really multithreading

t1 → [t1,0 t1,1] → actually [1] thread. → executes serially.

TRUE → enables nested parallism, makes actually a new team of threads.

if FALSE

to → to, t1 → to → to0,0 to0,1 to0,2 to0,3

t1 → t1 → t1,0 t1,1 t1,2 t1,3

illusion single thread.

this will also = 1

this will also executes = 1

if TRUE → then threads = 4

even if single (nested)

Environment variables

NO:

DATE: 10 March 2025

at runtime

void omp_set_dynamic (int flag)

↳called from outside of parallel region

omp_get_max_threads()

omp_get_num_procs()

↳ # of processors in system

⇒ omp_set_dynamic(1) → it will dynamically decide how many threads be created

↳true

omp single nowait

↳remove implicit barries (means other threads not going to wait for it)

not necessary 8 threads, can be minimum or more (not so much)

single thread will execute this line.

OMP_SCHEDULE → static, dynamic, guided.

↳chunk size

default=1

windows: $env: OMP_SCHEDULE = "guided, 4"

ubuntu: export OMP_SCHEDULE = " "

NO:
DATE:

Message passing :-

- synchronous (indication from server if it's ready)
- asynchronous (no need to wait)
  - buffers
  - faster plus less reliable.

↓request / response

Communication Model :-

$\frac{d}{s}$ = dprop dproc

$\frac{L}{R}$ = dtrans dproc

th → time for each hop

n = # of hops

Architectures :-

- P2P → Napster, Gnutella,
  - to download music (mp3) file sharing
- client / Server

eg

Issues :

- Programming complexity
- Scarce Robustness

n2 → n2
n1 → n5 } same type

NO:
DATE:

17 March 2025

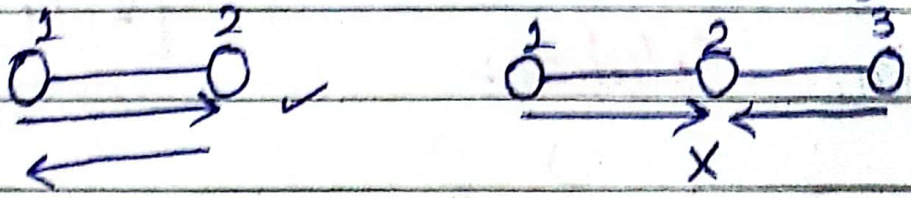
Basic Communication Operations

Assumptions:-

1) support cut-through routing
   - message should be able to transfer from router even if one bit of a packet is received

2) links are bi-directional
   - data sending can be on both ways at one time or at one single clock cycle

3) Single-port
   - In one clock cycle
     - one place receive or send at single time.



m = size of message tw = trans time for one word
ts = startup time

$\frac{mts}{100MB * \frac{1}{10Mbps}} = 10s$ → transmission delay.

NO:

DATE: 19th March 2025

circular 5-shift

$q \bmod \sqrt{p}$

→ # of shifts = 5

# of nodes

$p = 16$

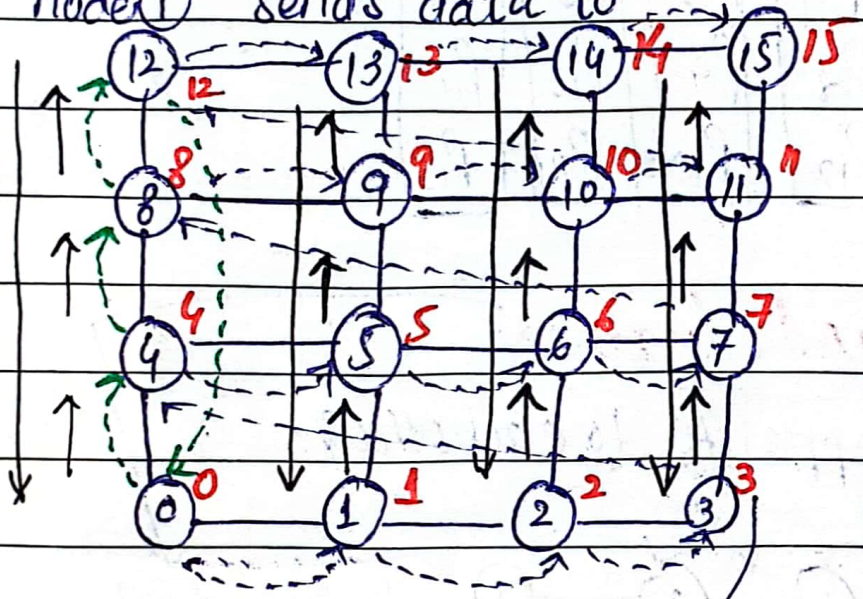
# of rows & cols = $\sqrt{p}$ = 4

4x4 mesh.

$\sqrt{p} * \sqrt{p}$

circular shift → naive

node (i) sends data to



* all links are bidirectional

(t1)

① right shift all

(t2)

② one shift up of left most column.

↓ manage mod stuff.

(t3)

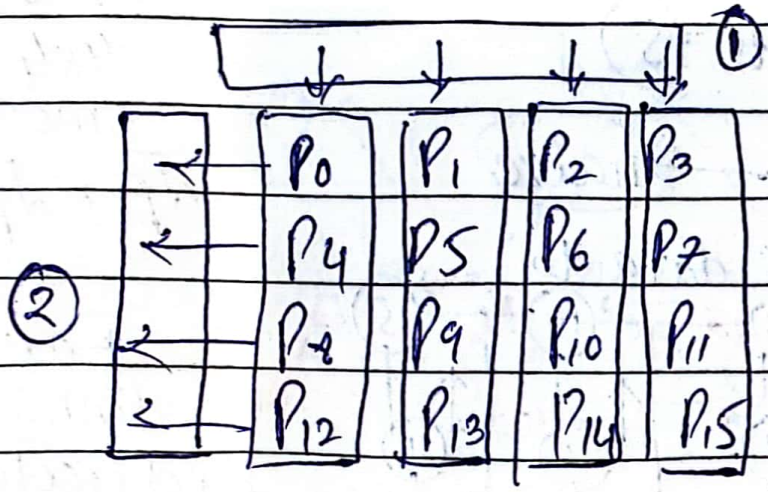
③ each one shift up

These values shift.

NO:
DATE:

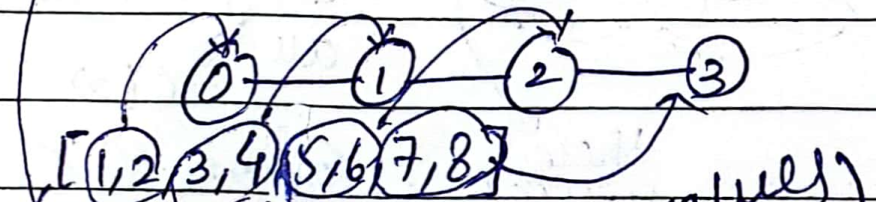
⇒ Matrix Vector Multiplication :-

1) One to All broadcast — log p
2) All to One reduction.



Scatter and Gather

opposite to each other.



send mutually exclusive values to separate nodes.

(2 values each node.

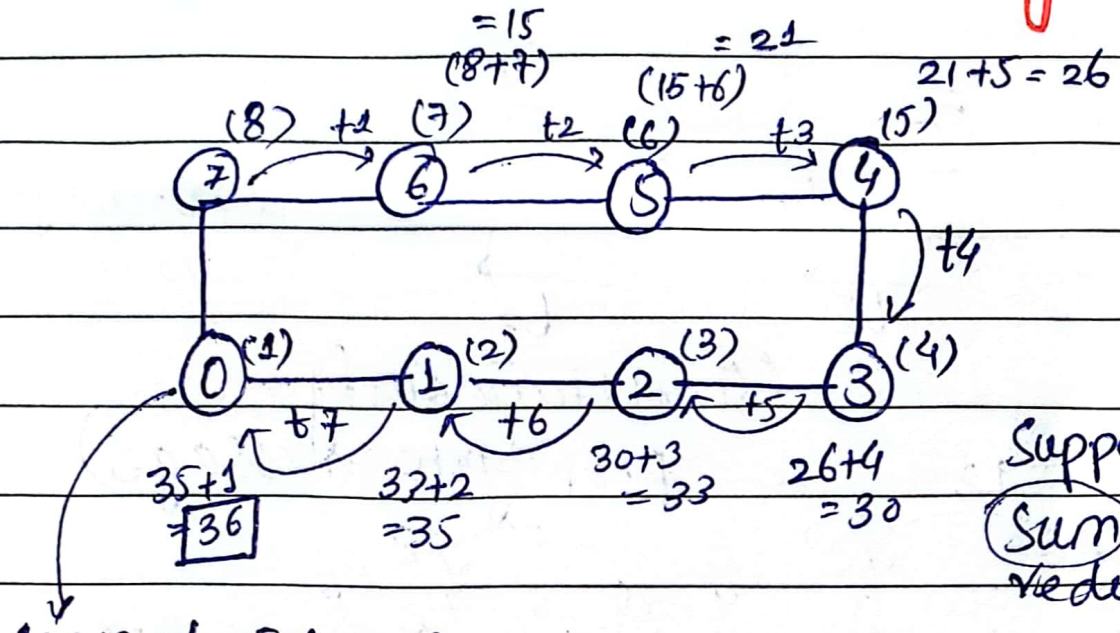
One node may have 8 values.

NO:

DATE:

• naive (p-1)

Reduction: • recursive doubling ($\log_2 p$)



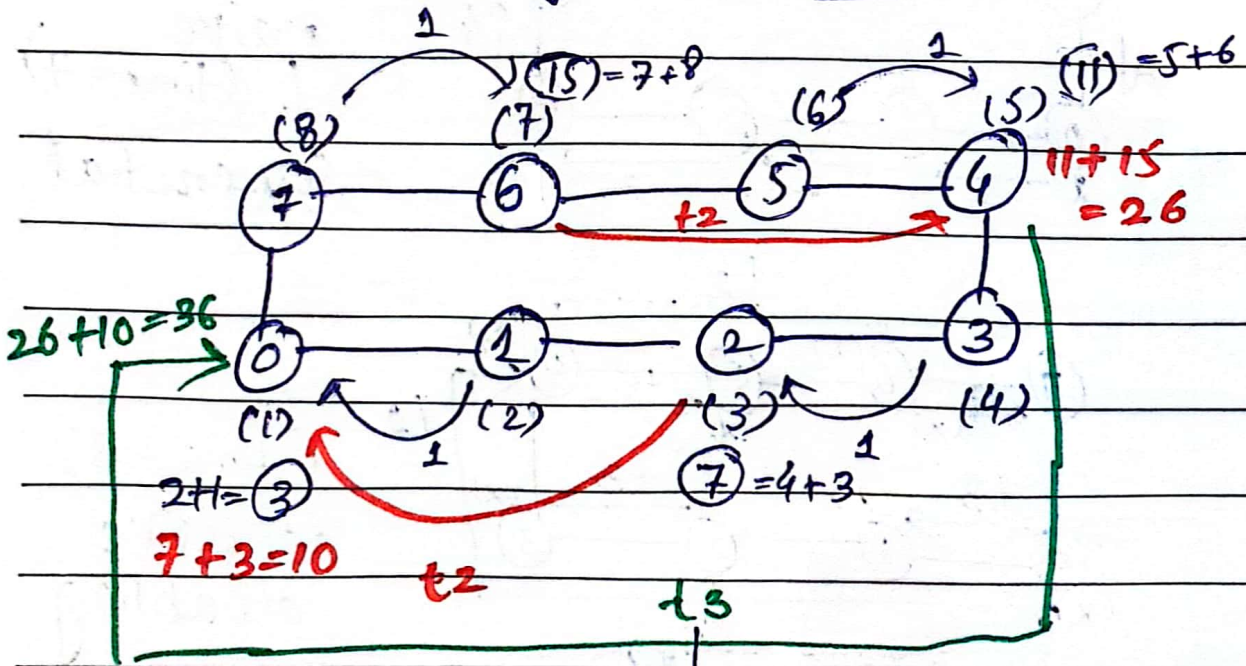
Suppose = Sum and reduction

1+2+3+4+5+6+7+8

7 clockcycles → naive :-

opposite of broadcast

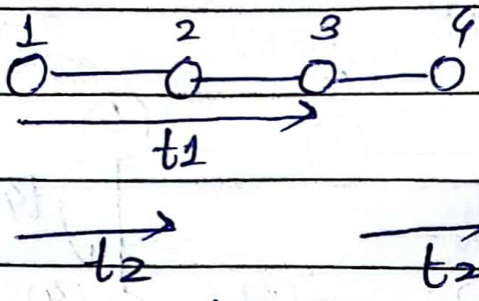
recursive doubling :- → one to all all to one same time.



$\log_2 p = \log_2 8 = \boxed{3}$

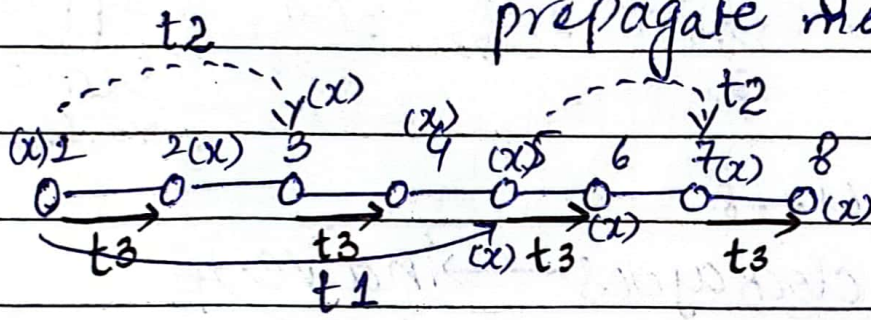
NO:
DATE:

Recursive doubling :-



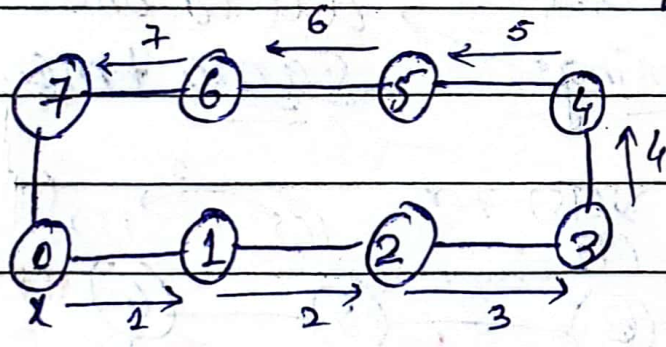
2 clock cycles now.

(Simultaneously propagate message)

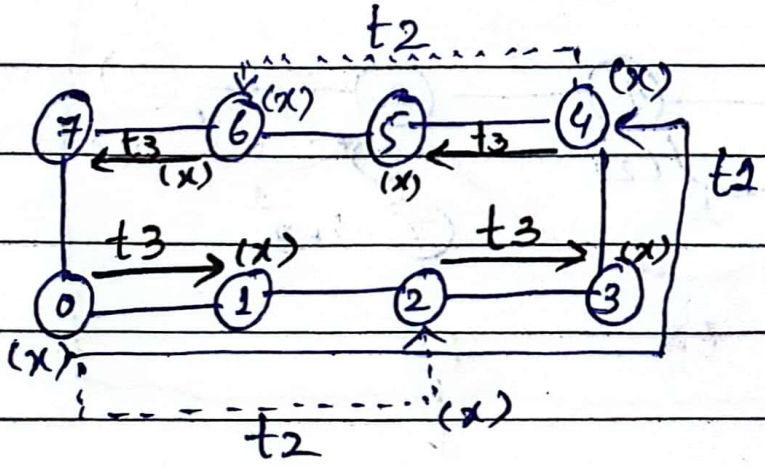


naive
↳ 7 clock cycles.

$\Rightarrow \log_2 p = \log 8$
$= \boxed{3}$ clockcycles.



naive
↓ (time = 7)
Sequential



(time = 3)
recursive doubling

$\boxed{\log_2 p}$

NO:

DATE:

One to All Broadcast :-

↳ single process sends identical data to all other processes.

each have copy of size m

All to One → opposite of broadcast

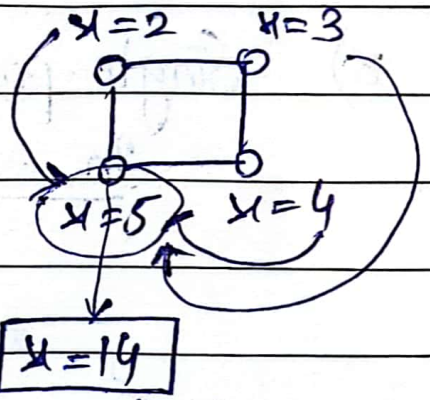
Reduction :-

All processes data are accumulated at single destination.

into one buffer of size m.

↳ through associative operator

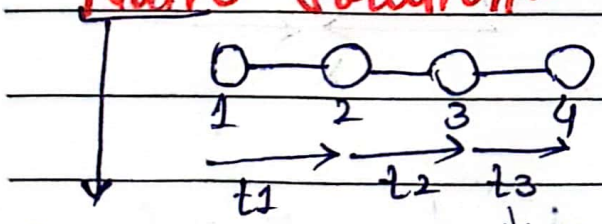
↳ +, -, *, ...



e.g sum.

Linear Array or Ring:

Naive Solution:-



↓ clock cycle 3

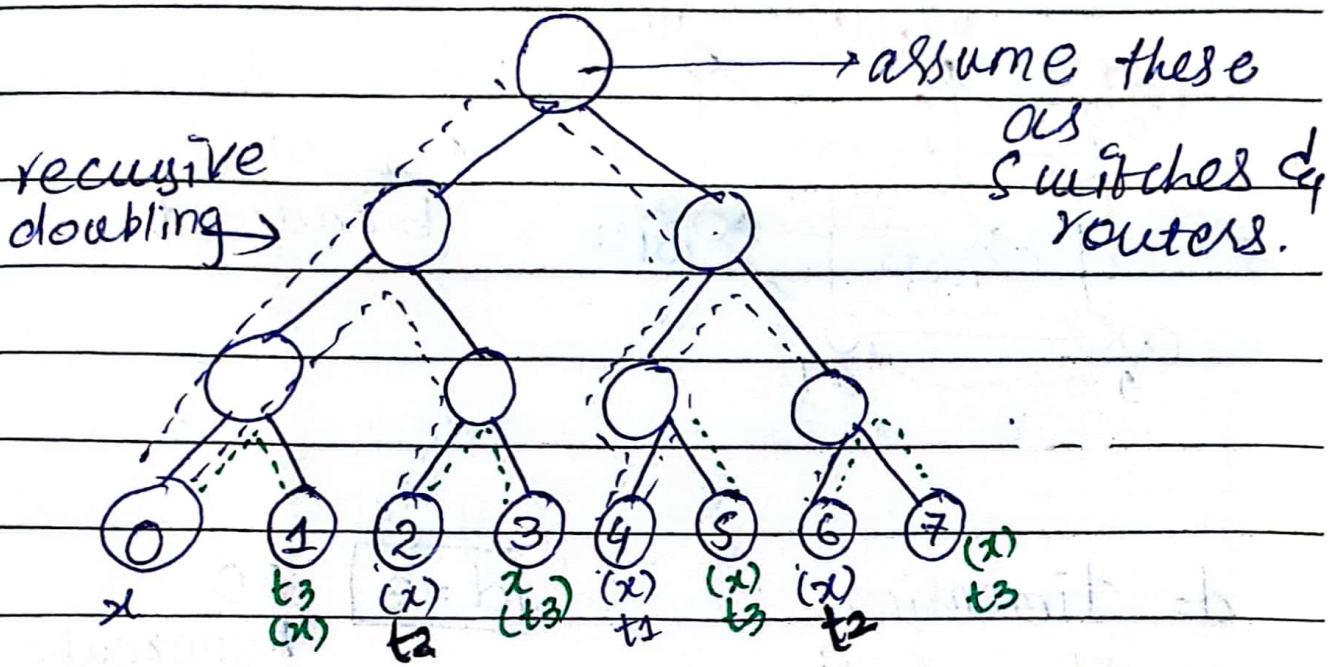
Sequential message transfer

p - 1 message

↳ clock cycle for p processes.

NO:
DATE:

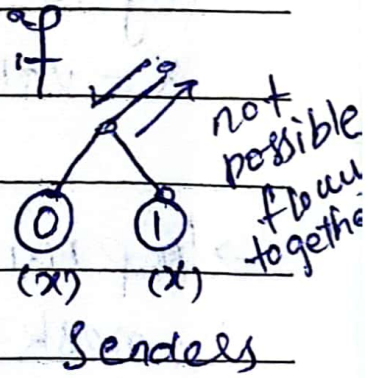
Binary tree One to All Broadcast



(3) clockcycles

$\log 8 = 3$

$\boxed{\log p}$



All to One Reduction

opposite

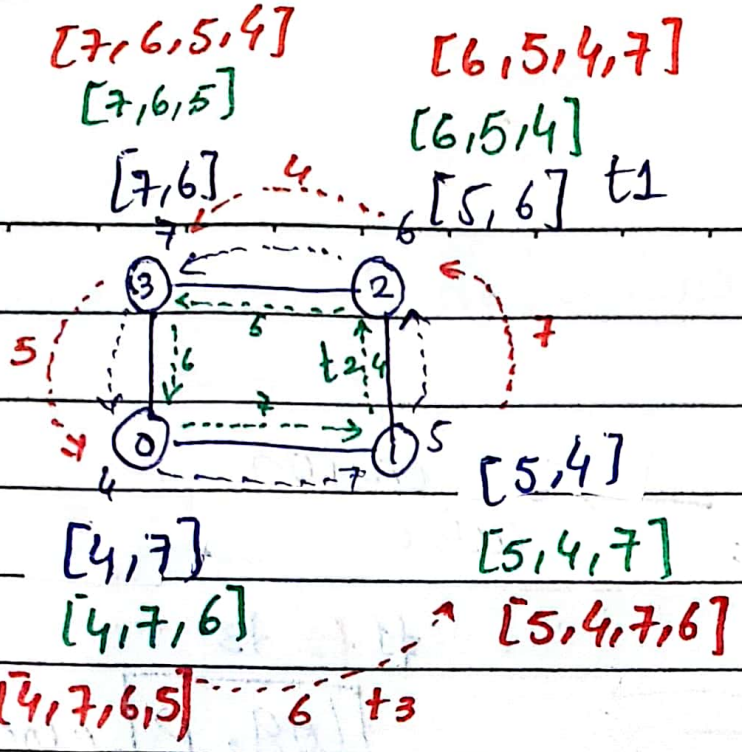
① every leaf send to its left neighbour

(1, 3, 5, 7)

② (6, 4, 3, 0) (odd values)

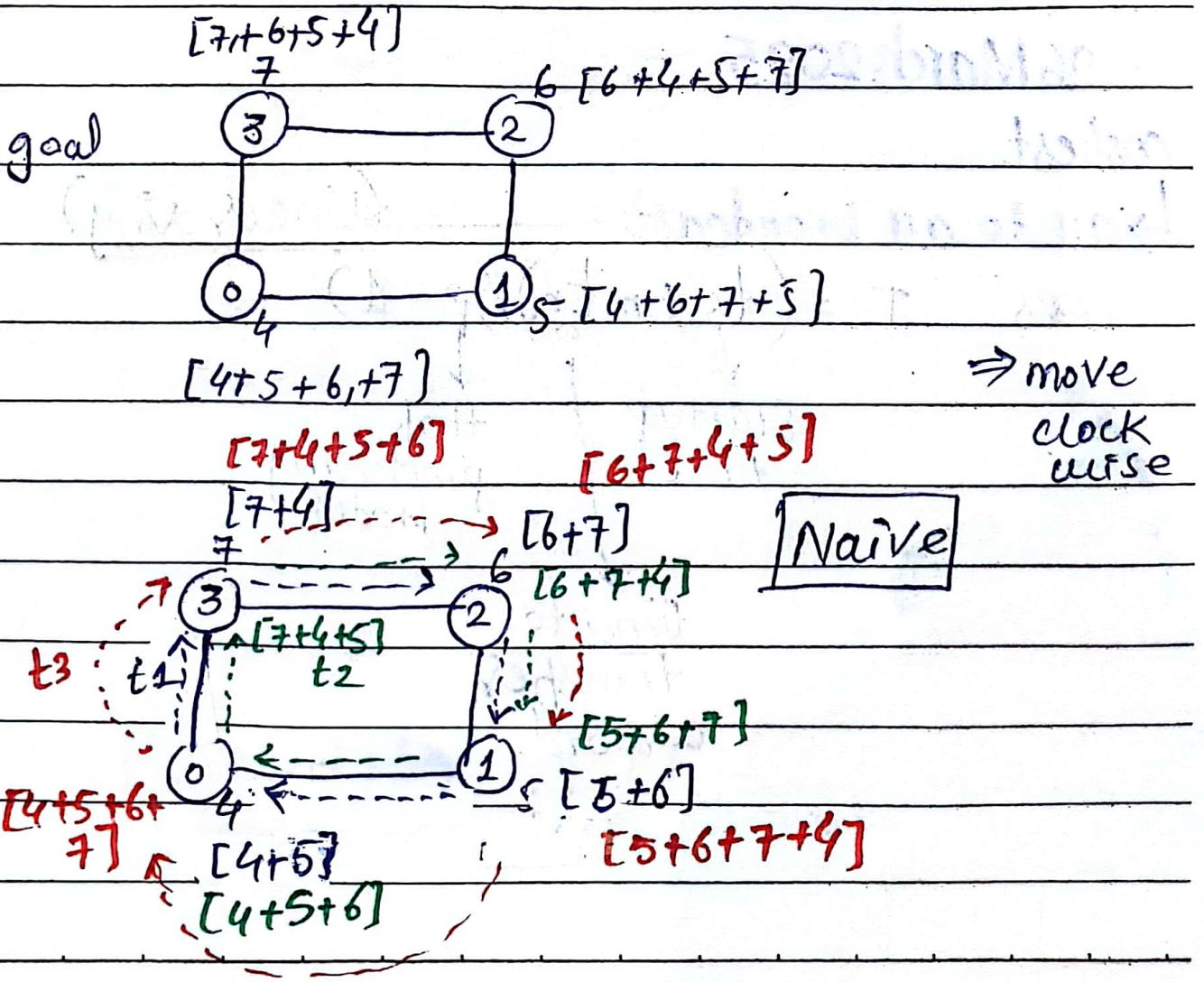
③ 4, 0

NO:
DATE:



associative operation

All to All Redution :→ every node is applying all to all redution.



NO:

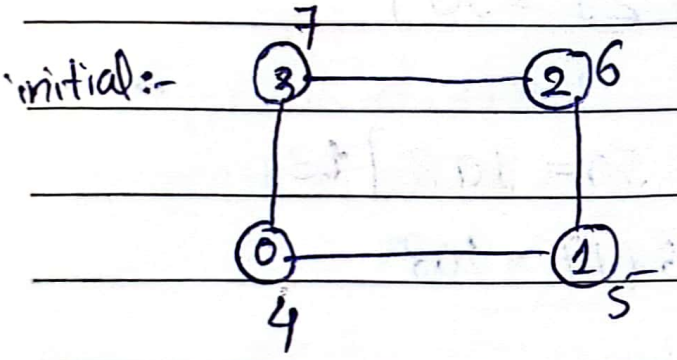
DATE:

dual → opposite.

Scatter: 1 node has several values So, send some of these values to some nodes

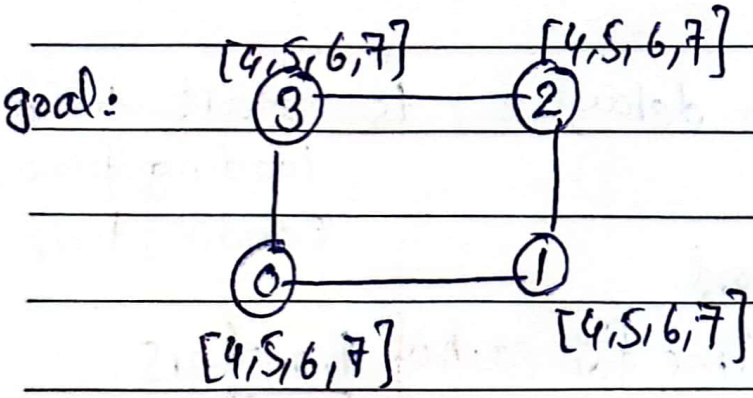
Gather: It is opp of gather. But we are not applying any associate operations

all to all broadcast :-



→ Recursive doubling better then naive

but in this case Naive modification is better

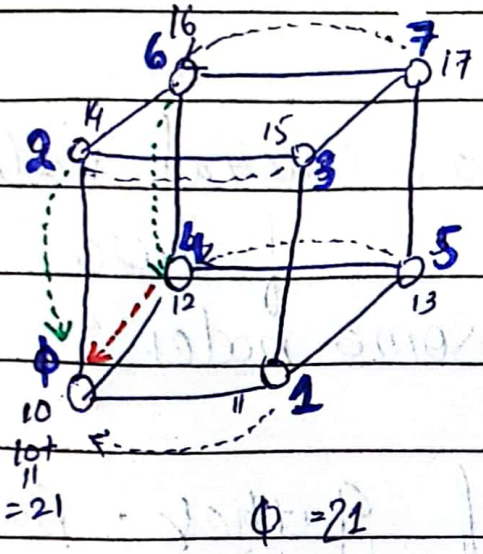


NO:

DATE:

24 March 2025

Topology :- linear array, 2-D ring, Hypercube, 2D-mesh.



all to 1 reduction (associative operation)

$\phi = 21$

$4 = 13 + 12 = 25$

$2 = 15 + 14 = 29$

$6 = 17 + 16 = 33$] t1

$4 = 33 + 25 = 58$

$\phi = 29 + 21 = 50$] t2

$\phi = 58 + 50 = 108$] t3

$10 + 11 + 12 + 13 + 14 + 15 + 16 + 17 = 108$

↓ to check correction.

$m t_w$ = transmission delay

↓ size of message → time to transmit one word.

t_s = start time / loading time / sending time

$t_s + m t_w$ → time for each of transfers.

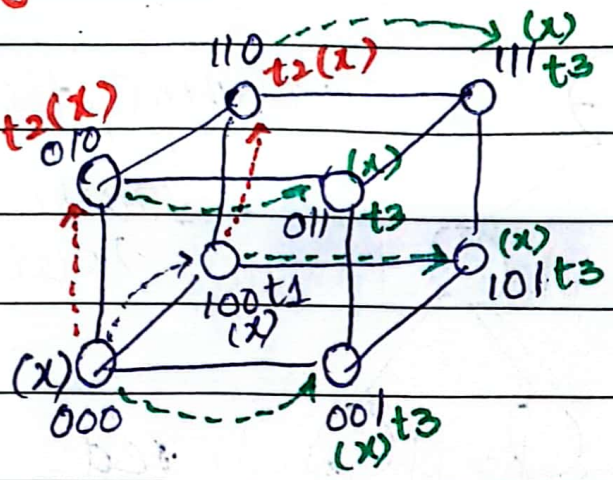
$T = (t_s + m t_w) \log p$ → Cost

NO:

DATE:

Hypercube:-

p = 8 nodes



1-to all broadcast

d = dimension

my-id = node-id

x = value

$\boxed{d = 3}$ 3D hypercube

$mask = 2^3 - 1 = \boxed{7}$

$(7)_{10} = (111)_2$

$\boxed{i = 2 = d-1}$

→ loop runs d times

mask = mask XOR 2^i

(111) XOR 2^2 (100)

mask = $\boxed{011}$

→ to decide senders or receiver.

my id AND mask

if (000 AND 011) = 000

if (my id and 2^2)

000 AND 100 = 000

1 node ← All-to-1 Reduce

1 to All Broadcast

NO:

DATE:

All-Reduce: → MPI All Reduce.

All to All Reduction: → Same like

Strategies:

M#1) Use all to one reduction followed by one to all broadcast.

M#2) step by step sharing of msg in dimension (1, 2, 3) replace union with associative operator.

Prefix-Sum: → like cummulative sum

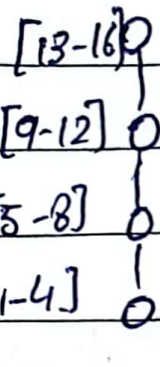
| 1 | 2 | 3 | 4 | 5 | 6 |

$\theta(n)$

| 1 | 3 | 6 | 10 | 15 | 21 |

NO:

DATE:



in column four array is shared rather the value.

add ① & ②

Total time = $\boxed{2t_s(\sqrt{p}-1) + mt_w(p-1)}$

all to all broadcast clockcycle → $\boxed{\log_2 p}$

hypercube → dimension wise

cost est.

1st step = $(t_s + mt_w)$ → 1 value swap (shared)

2nd step = $(t_s + 2mt_w)$ → 2 values swap

ith step = $(t_s + 2^{i-1} m t_w)$

Total cost = $\boxed{\sum_{i=1}^{\log(p)} (t_s + 2^{i-1} m t_w)}$

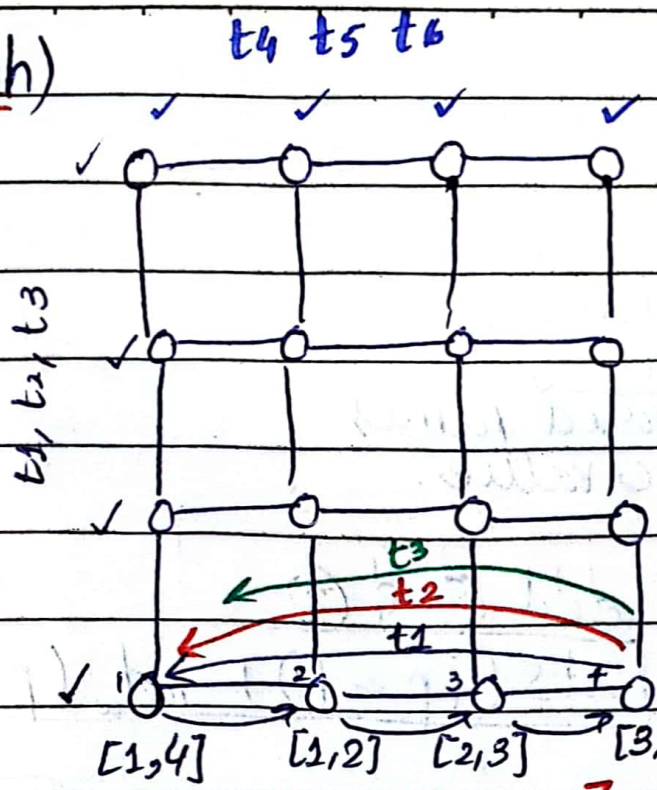
Simplify → $T = \boxed{t_s \log p + m t_w (p-1)}$

costest → all to all reduction

NO:

DATE:

(Mesh)



done in all rows.

rowwise ①

[1,4,3] [2,1,4] [3,2,1] [4,3,2]

[4,3,2,1]

[1,4,3,2] [2,1,4,3] [3,2,1,4]

② columnwise

$p = 16 \quad \sqrt{p} = 4$ # of rows # of cols.

$T(\underline{\text{first phase}}) = (ts + mtw)(\sqrt{p} - 1)$ — ①

↓ rowwise

↓ # of cols

$T(\underline{\text{second phase}}) = (ts + (\sqrt{p})\, mtw)(\sqrt{p} - 1)$

columnwise

└ ②

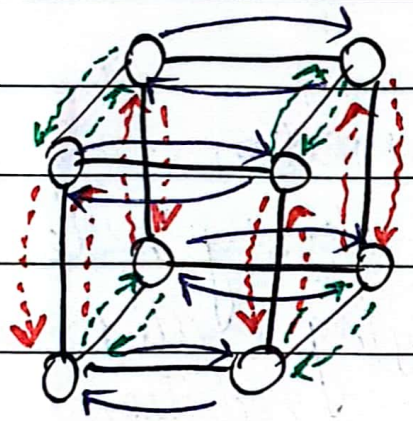
↓ every node share an array rather than number

NO:

DATE:

⇒ all to all broadcast on 2D mesh

Hypercube: → dimension wise



+1 +2 +3

$\boxed{\log_2 p}$ clockcycle

cost

26 March 2025

costest

↳ all to all broadcast ——— (Linear ring)

$$T = (t_s + m t_w)(p-1)$$

- t_s: startup time
- p: # of processing nodes.
- t_w: time to transfer a word.

theoretical → no parallel overhead
experimental → "

NO:
DATE:

cost analysis: $\sqrt{p}$ processors.

$$(t_s + (t_w m P/2)(\sqrt{p} - 1)$$

t_w ↓ transmission time for word.

⇒ Hypercube:

OpenMP

NO:

DATE:

$m(p-1)$ } size of message — # of clock cycles = $p-1$

$(p-1)$ ← # of unique messages.

All to All Personalized [Ring].

Cost Analysis: (size decrease at each t)

$$T = \sum_{i=1}^{(p-1)} (t_s + (p-i)\, m\, t_w)$$

T ← total time, communication cost

$=$ Simplify

$$= (t_s + \left(\frac{1}{2}\right) p m t_w)(p-1)$$

⇒ mesh:

0 → 1 → 2 ← ① row level done

② column level propagate

NO:

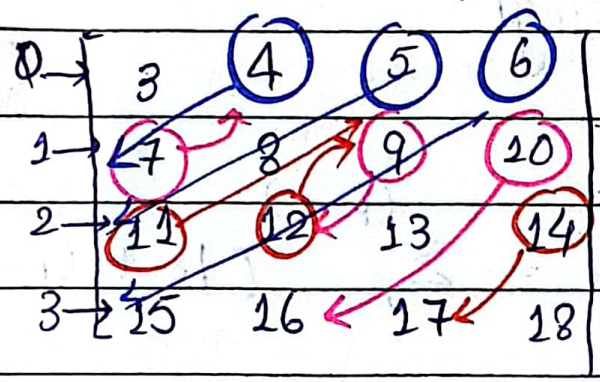
DATE:

7th April 2025

All to All Personalized → all nodes do scatter and it's happening in parallel

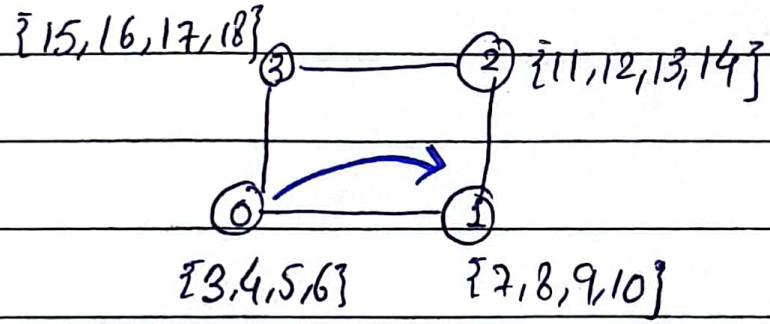
M (0), (p-1) → destination

source



diagonal don't move

Transpose:



in t1, it will (R1) keep 4 and pass the rest

{4[0,1], 5[0,2], 6[0,3]}

S D

Simultaneously in parallel

{7[1,0], 9[1,2], 10[1,3]}

⋮

{11[2,0], 12[2,1], 14[2,3]}

p-1 → p-2 → p-3 → p-4

at every clock cycle, size of message decrease by 1

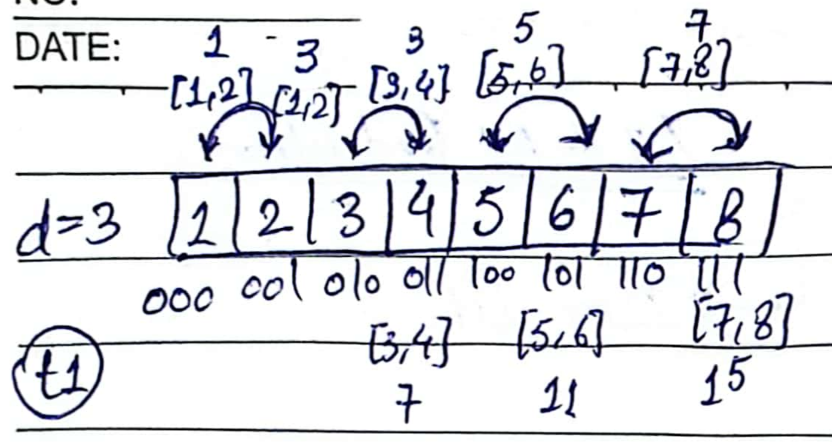
m(p-1) → m(3)

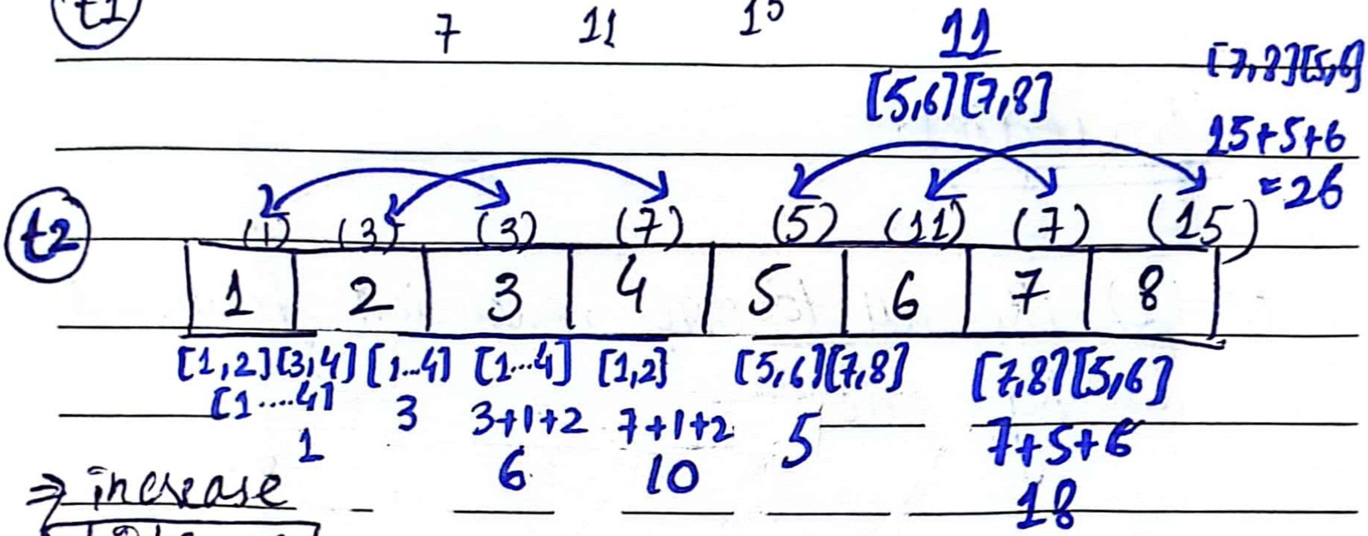
unique message

NO:

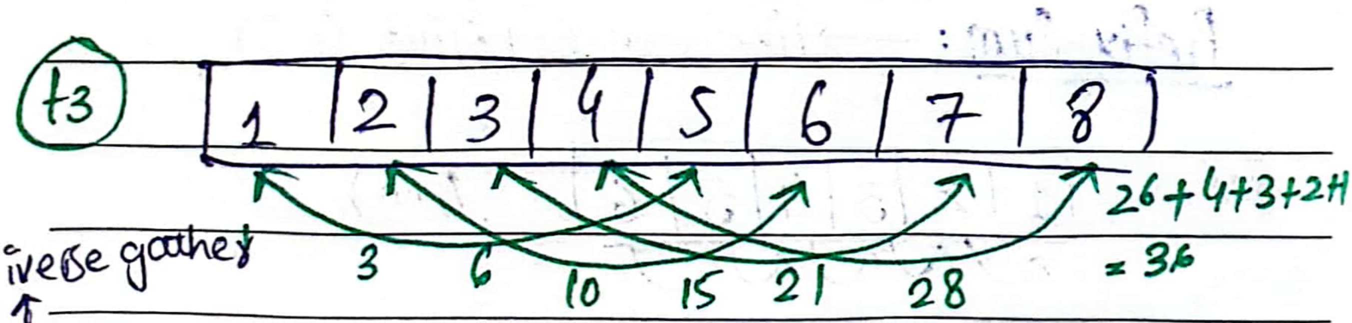
DATE:

Prefix Sum hypercube.





→ increase distance of neighbour in power of 2 at every clockcycle



inverse gather

↑

= 18+1+2+3+4
= 28

Scatter: