Types of Parallel Computers

Flynn's classical Taxonomy

SISD **Single Instruction, Single Data**	**SIMD** **Single Instruction, Multiple Data**
MISD **Multiple Instruction, Single Data**	**MIMD** **Multiple Instruction, Multiple Data**

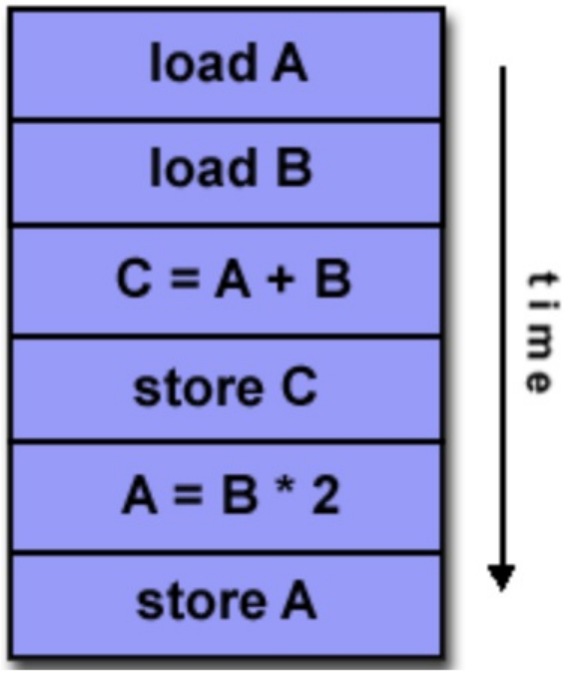
Instruction Set
Data Stream

Serial computer

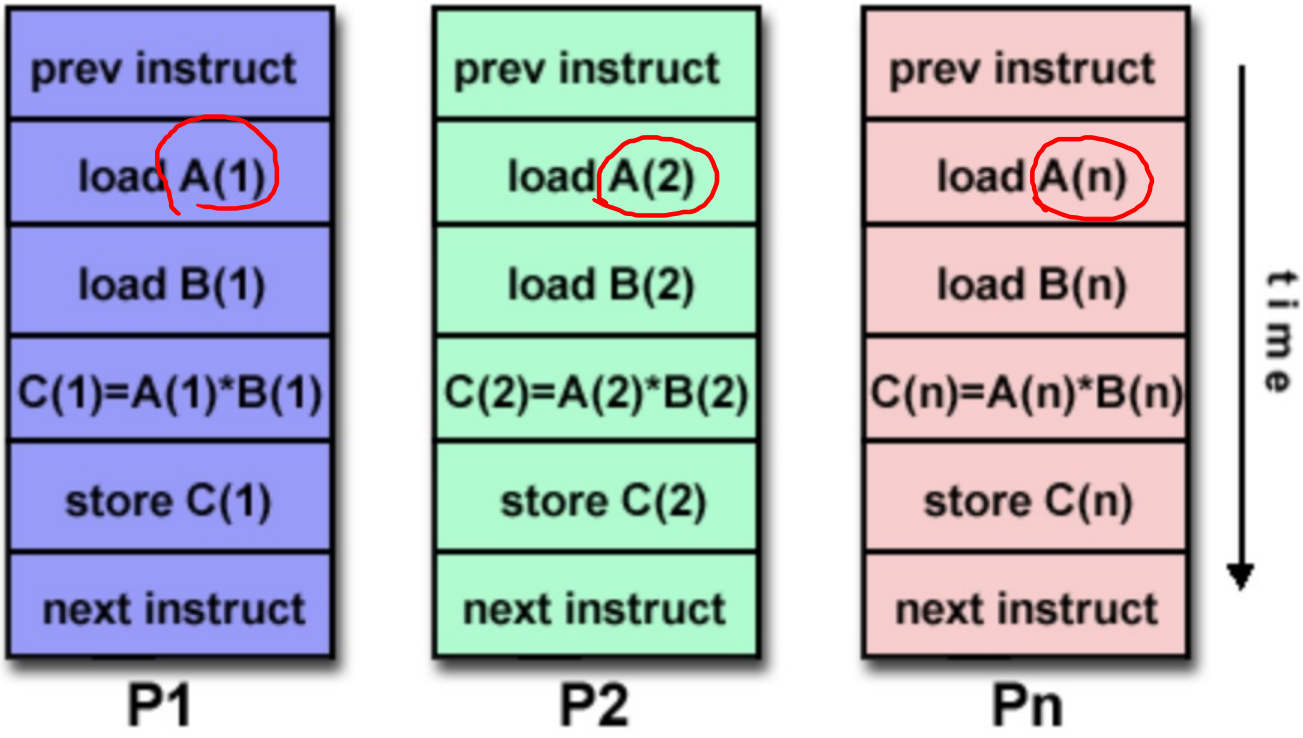
Parallel

S

SISD

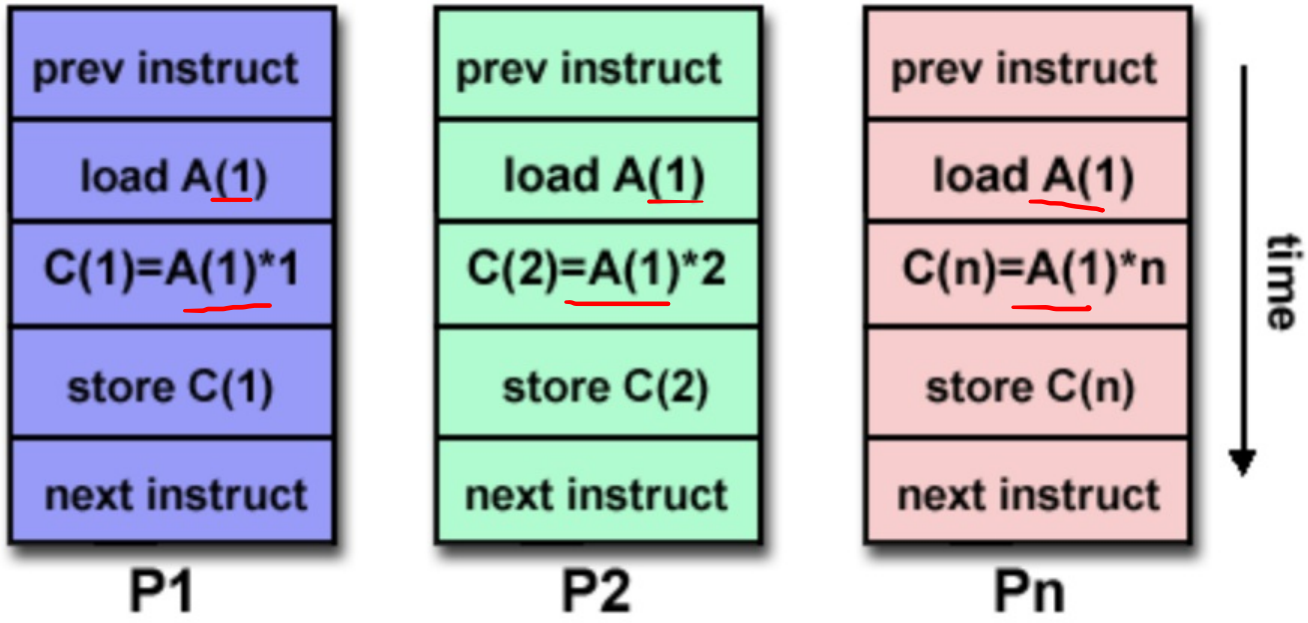


SIMD

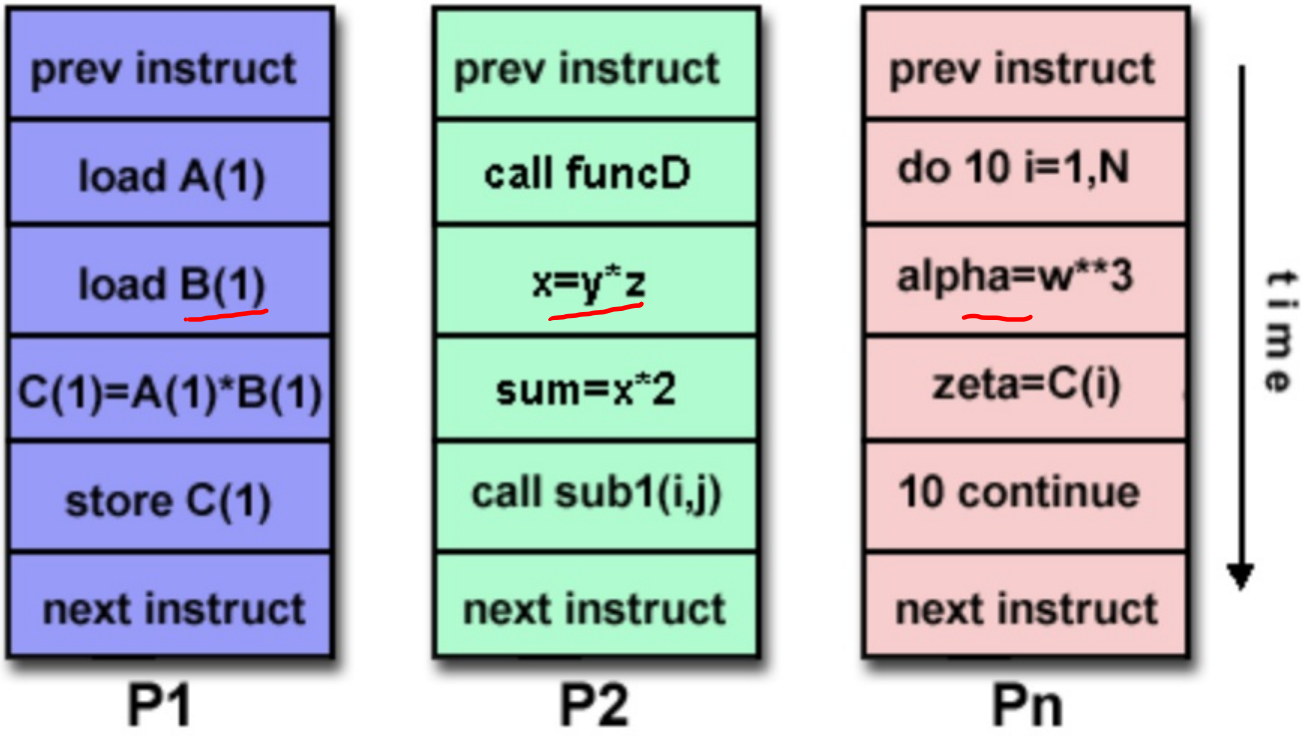

prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
load A(2)
load B(2)
C(2)=A(2)*B(2)
store C(2)
next instruct
P2
prev instruct
load A(n)
load B(n)
C(n)=A(n)*B(n)
store C(n)
next instruct
Pn
time

MISD

Research



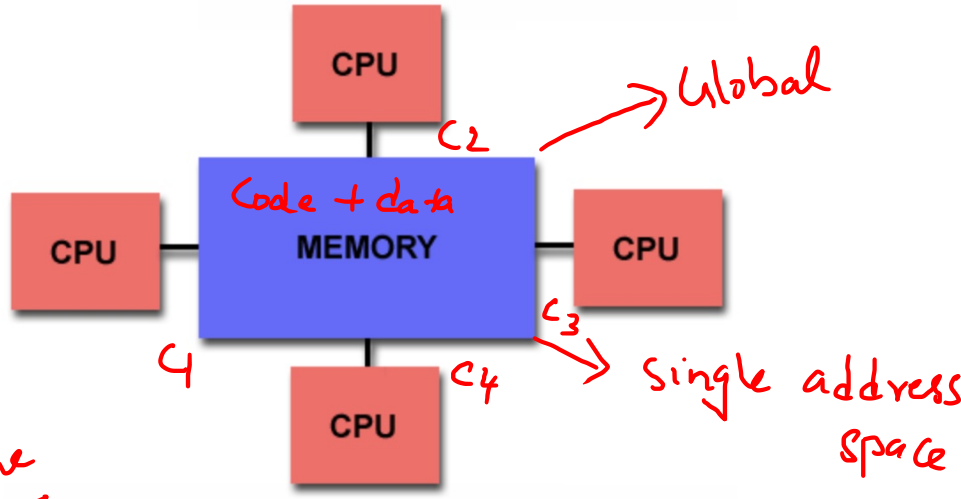
MIMD


prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
call funcD
x=y*z
sum=x*2
call sub1(i,j)
next instruct
P2
prev instruct
do 10 i=1,N
alpha=w**3
zeta=C(i)
10 continue
next instruct
Pn
time

m/y Architectures.

1. Shared

Executable code



Two forms

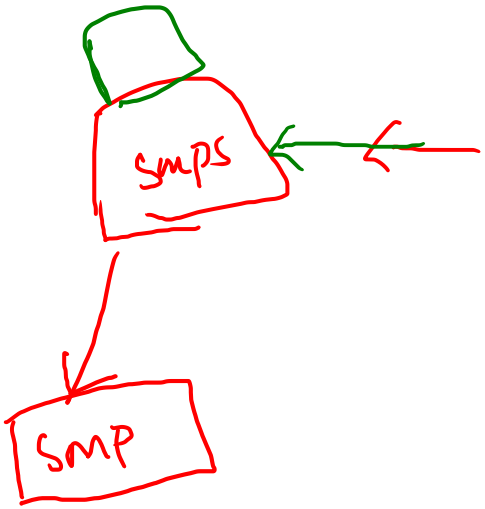
(1) Uniform m/y access (UMA)

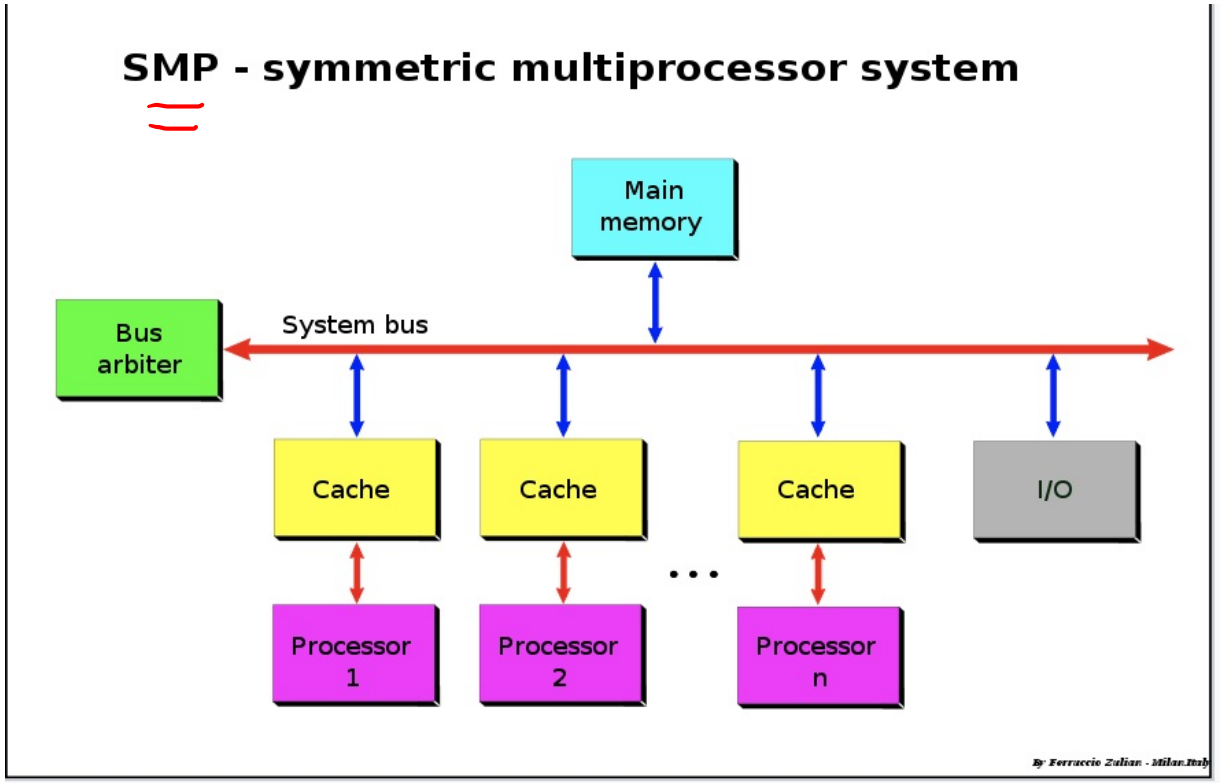
(2) non-uniform m/y access (NUMA)

Cache → preserving locality

CC-UMA

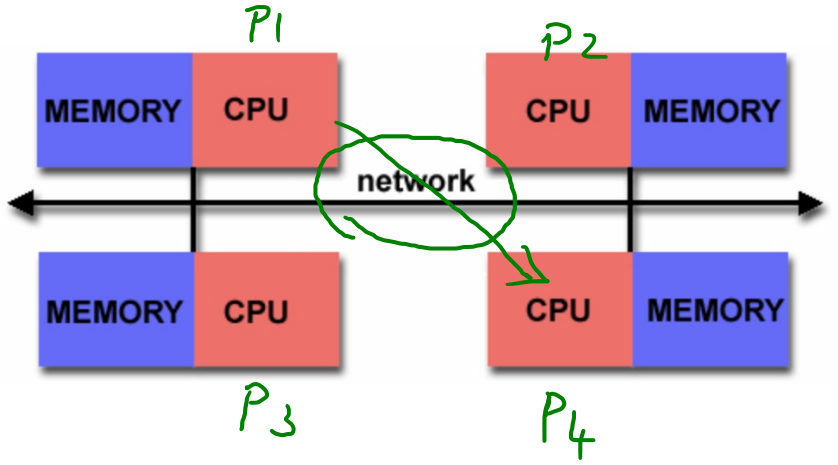
Cache coherent





a processor can access its own local m/y faster than non-local m/y

Distributed Memory



Ethernet

m/y address
↳ no concept of global address space.

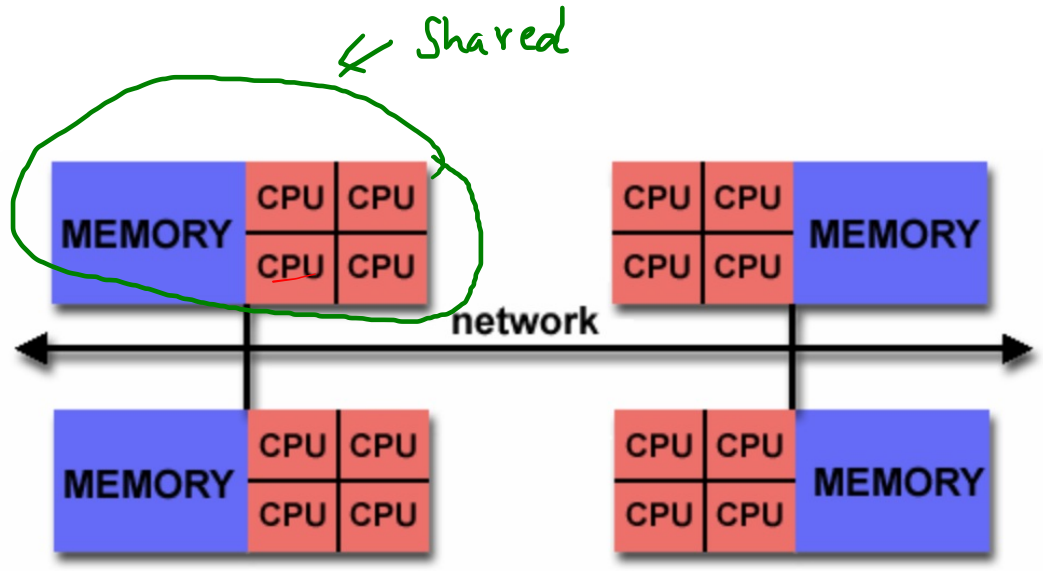
Concept of cache coherency ? {not applicable}

message passing ⟶ Programming

MPI ≔

Hybrid models

Shared +
distributed



Types of pavalled computers

1. Shared
2. Distributed
3. Hybrid.

Parallel Programming Models & Paradigms

Traditional — sequential (cost of components)

↓ high availibility

Parallel pradigms

↓

{Cluster} → cost-effectiven

n/ws

↓

Super computers

Scalability

Scalable computing clusters ?

↓

{PC or WS or SMPs.}

↓

homogeneous

hetereogeneous

APC — NASA

↳ High Performance computing

Allocation powers

Intelligent mechanisms.

(huge data set)

n/w wide resource sharing

↳ Scalability performances

↳ flexible use of WS

↳ multi user - mgmt →

Main motivation

- minimize economic risk
- Consumer off-the-shelf technology

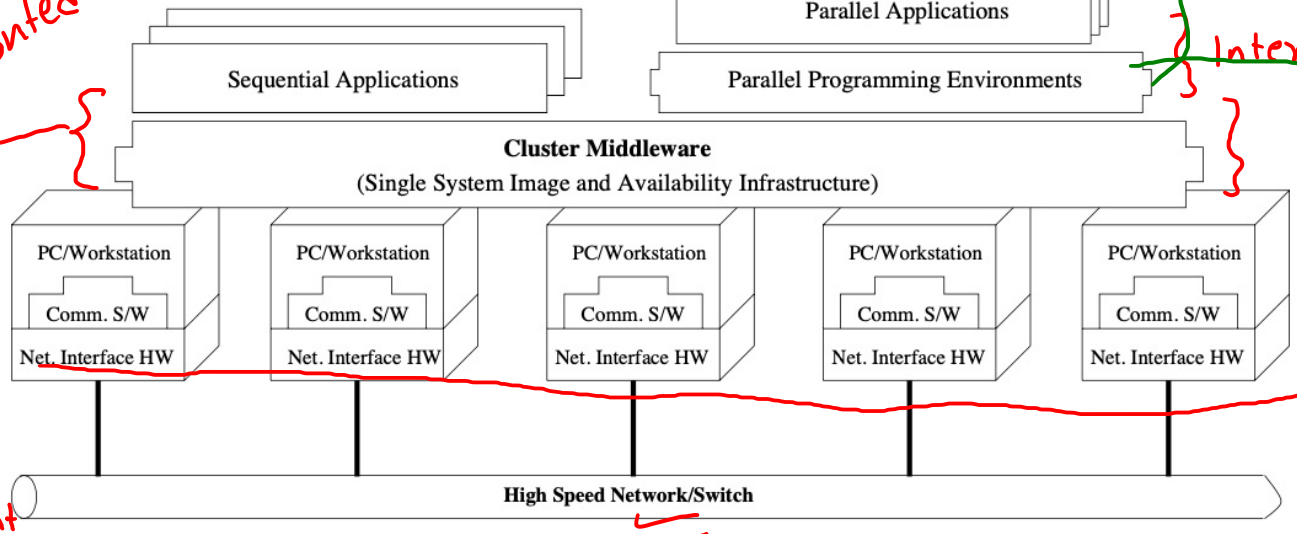
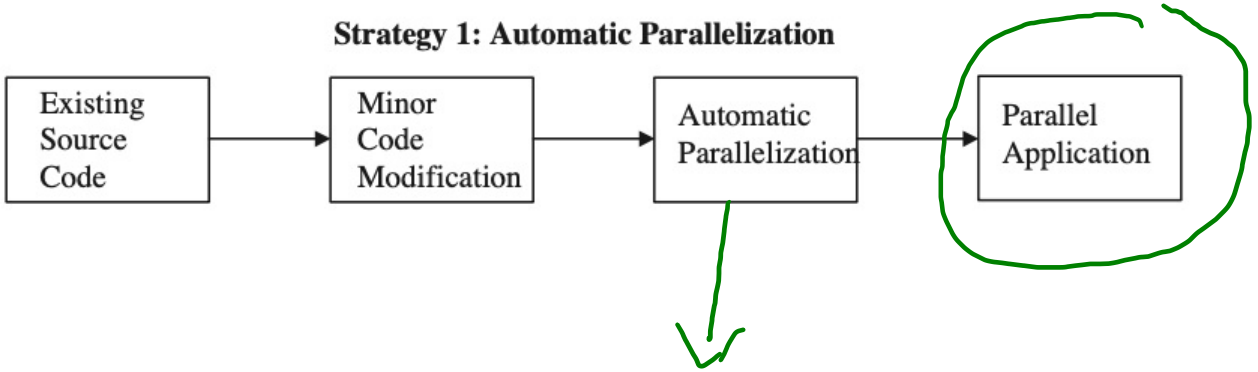


Figure 1.1 Cluster computer architecture.

Strategies for developing parallel Apps

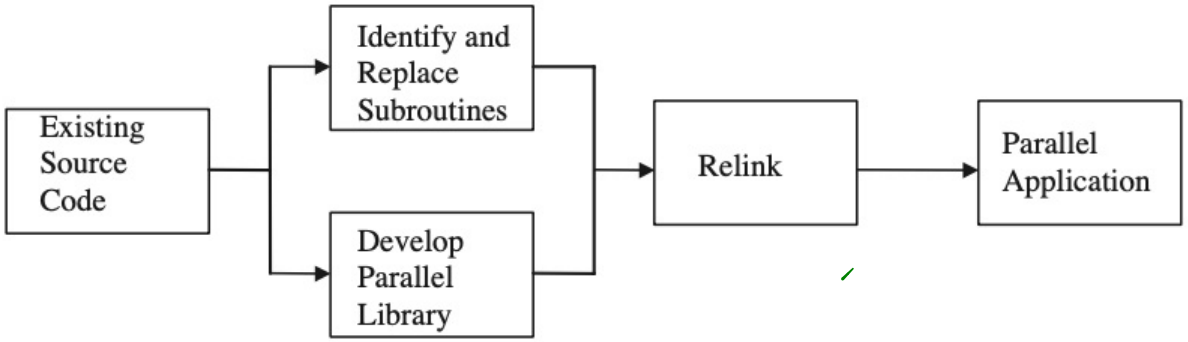


{ Compiler technology }

↳ recent

Successful

Strategy 2: Parallel Libraries



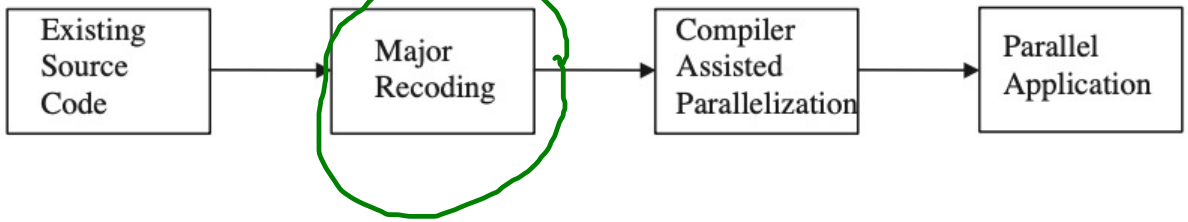
{reusability}

Parallel Libraries

(1) Control structures

(2) provide parallel impl^n of mathematical routines

Strategy 3: Major Recoding



Scratch

Level of parallelism { H/w, S/w }

Code granularity

1. Signal } H/w
2. Circuit }
3. Component } S/w
4. S/m level }

Code granularity → (lumps of code) grain size → (latency)

Code granularity

			Parallelised by
(1)	Very fine	→ multiple inst^n issue	→ Processor
(2)	Fine	→ data level	→ Compiler
(3)	Medium	→ control level	→ Programmer
(4)	Large	→ task-level	→ "

Level of parallelism { H/w, S/w }

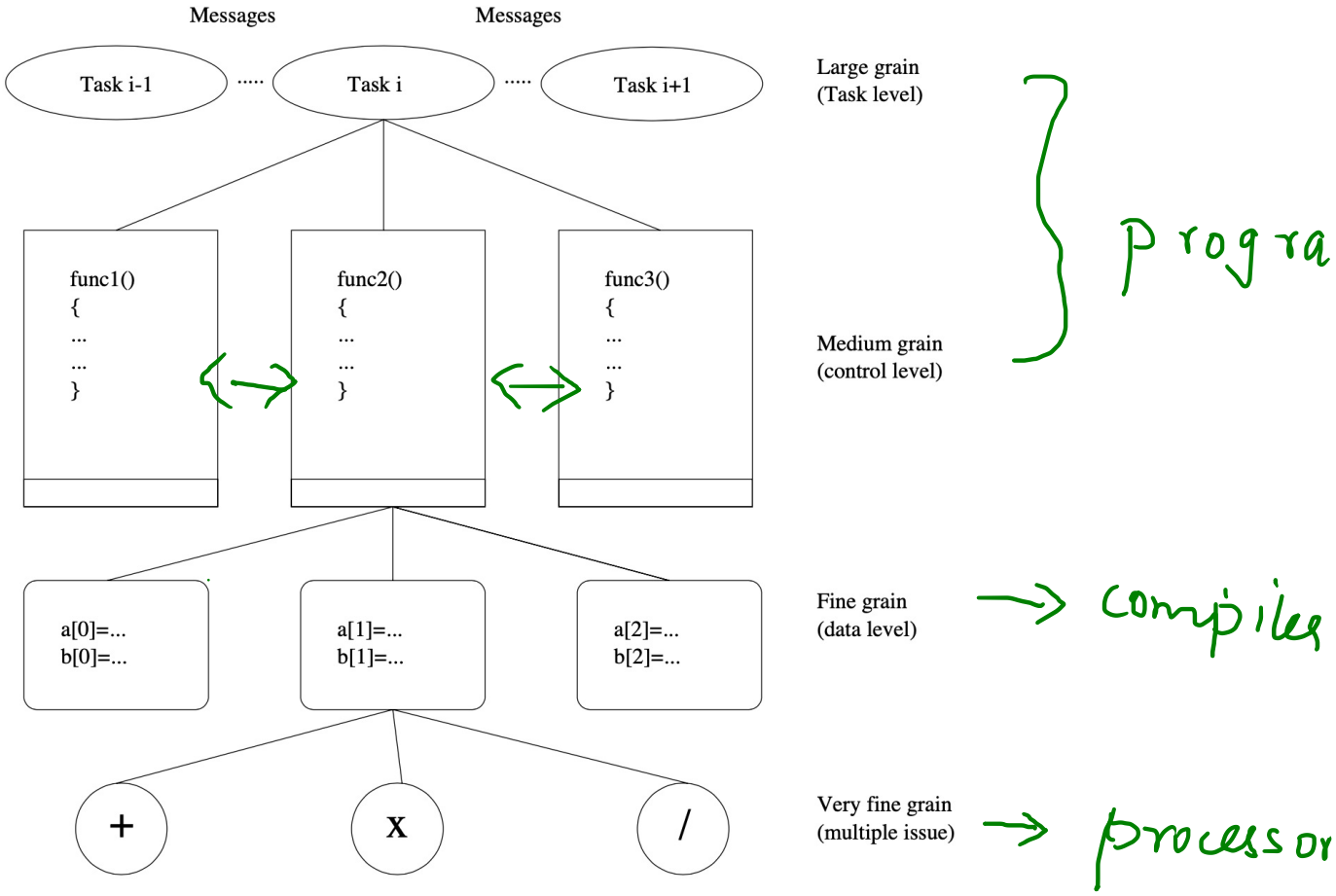
Code granularity

1. Signal } H/w
2. Circuit }
3. Component } S/w
4. S/m level }

Code granularity → (lumps of code) grain size → (latency)

Code granularity

				Parallelised by
(1)	Very fine	→	multiple inst^n issue	→ Processor
(2)	Fine	→	data level	→ Compiler
(3)	Medium	→	Control level	→ Programmer
(4)	Large	→	task - level	→ "


Messages
Messages
Task i-1
.....
Task i
.....
Task i+1
func1()
{
...
...
}
func2()
{
...
...
}
func3()
{
...
...
}
a[0]=...
b[0]=...
a[1]=...
b[1]=...
a[2]=...
b[2]=...
+
X
/
Large grain
(Task level)
Medium grain
(control level)
Fine grain
(data level)
Very fine grain
(multiple issue)
programmer
compiler
processor

Tools

1. Parallelizing Compilers ✓✓ 2 architectures
2. Parallel languages ✓✓
3. message passings → MPI ✓✓, PVM
4. Virtual shared m/y ← VM implements shared m/y pgmming model in a distributed m/y environment
5. Object Oriented pgmming
6. Pgmming skeleton.

MPI → Synchronization
- data partition
- map processes
- data structure i/p & op

data → smp

smp

Distributed Shared m/y

hybrid

(Data replication)
↳ back up

local

smp
data

Methodical Design of Parallel Algorithms

machine independent ✓✓ → early

machine - specific tasks → bit delayed

$\frac{\#\text{Computation}}{\#\text{comm}^{n}}$

(1) Partitioning → decomposing < data / functional

(2) Communication → Coordinate all the partitioned tasks

(3) Agglomeration → Evaluate first two stages

(4) Mapping → Load balancing

(4) maximizing processor utilization

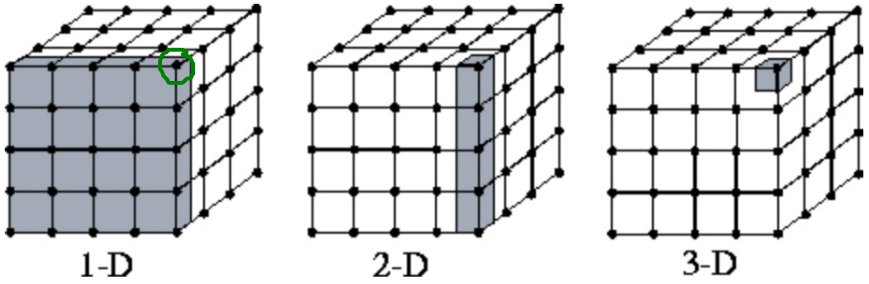
(6) Communication cost ↓

{ Scalability < h/w, s/w
Efficiency

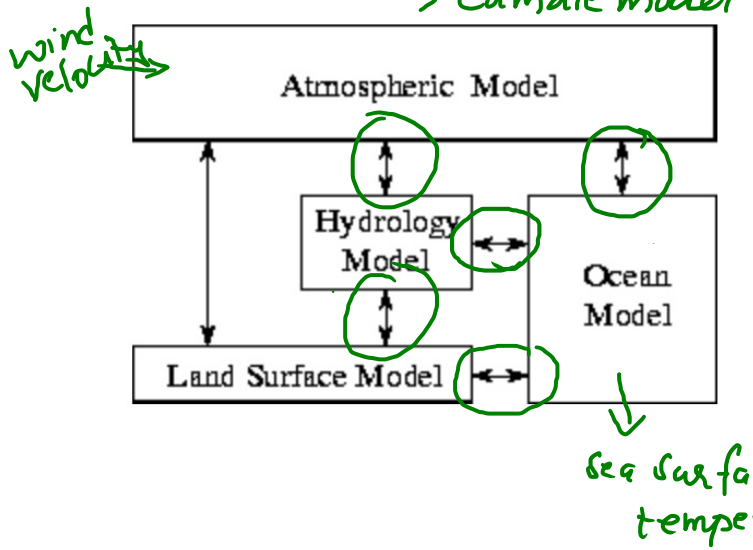
Methodical Design of Parallel Algorithms

1. Partitioning

Datastructure



Functional → climate model

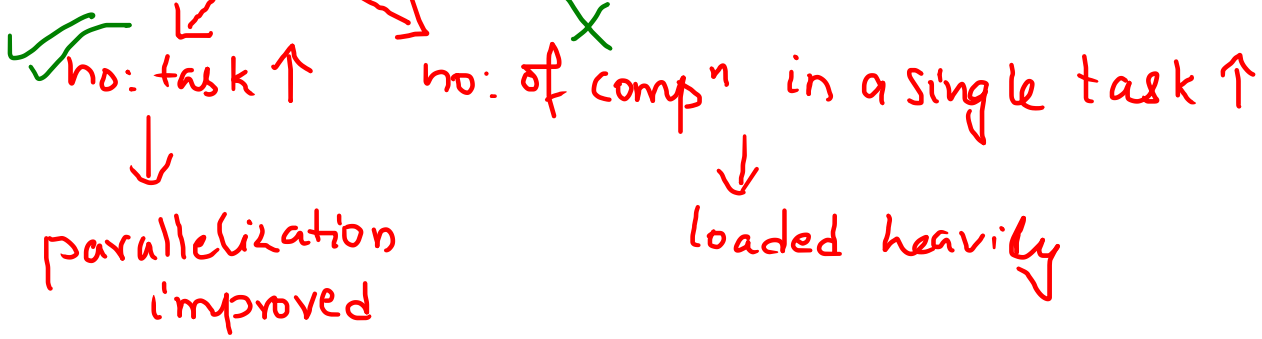


Decomposition → Very fine

{ Disjoint sets
  Overlapping

Computation — functional decomposition — Computations ← associate data
data — domain decomposition — first focus is on data — associate comput^{n} to data

1. n no: of processors → n no: of tasks
2. redundant computation / storage rqmts.
3. Granularity → Equal amount of work to processors
4. Problem size ↑ Does the no: of tasks ↑ or ↓ ?

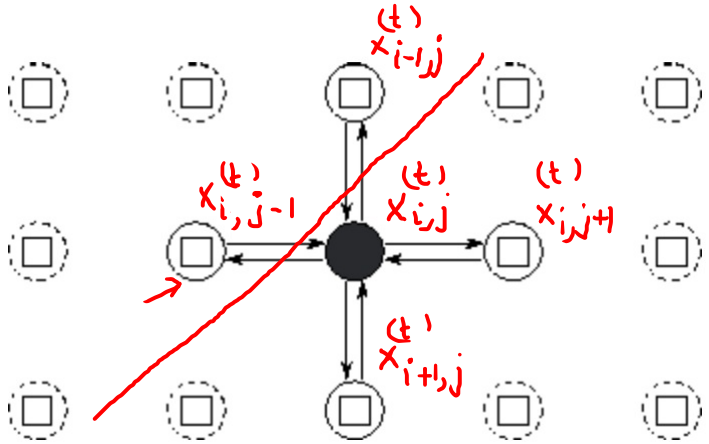


Methodical Design of Parallel Algorithms

Communication

1. Channel link
2. messages

computation / # communication



1. local / global comm^n ← small no: of other tasks; ← large no: of tasks
2. Structured / unstructured comm^n ← grid / tree; ← arbitrary graph
3. Static / dynamic comm^n ← set of tasks; ← varying over time.
4. Synchronous / asynchronous comm^n ← prod cons; ↓ Consumer

Domain

↓

{ Data disjoint set }

(Challenging)

Functional

↓

Data flow b/w tasks

$$x^{(t+1)}_{i,j} = \frac{4x^{(t)}_{i,j} + x^{(t)}_{i,j-1} + x^{(t)}_{i-1,j} + x^{(t)}_{i,j+1} + x^{(t)}_{i+1,j}}{8}$$

$\{ x_{ij}^{(1)}, x_{ij}^{(2)}, x_{ij}^{(3)} \ldots \}$

$\Downarrow r$

$x_{i-1,j}^{(1)}, x_{i-1,j}^{(2)} \ldots$

$x_{i+1,j}^{(1)} \ldots$

$x_{i,j-1}^{(1)} \ldots$

$x_{i,j+1}^{(1)} \ldots$

for $t = 0$ to $T-1$

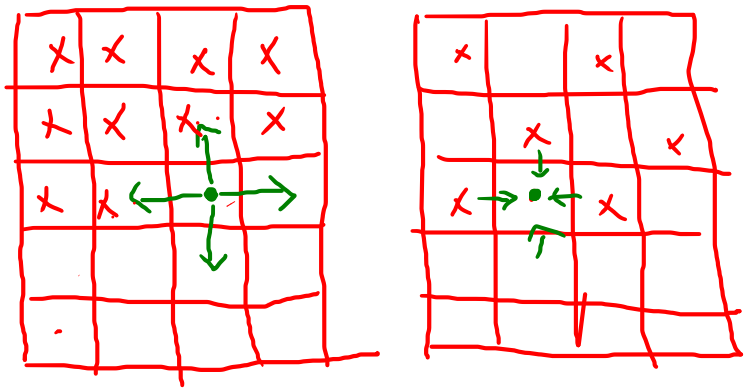
{

Send $x_{ij}^{(t)}$ to each neighbour

receive 4 values from neighbours

Compute $x_{i,j}^{(t+1)}$

}



Methodical Design of Parallel Algorithms

Centralized Summation

(1) Centralized algm

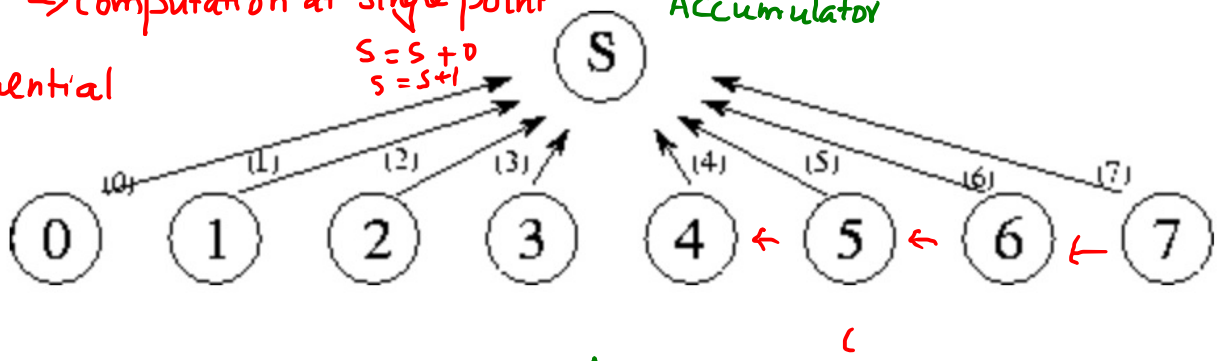
↳ Computation at Single point

Summation of array elements

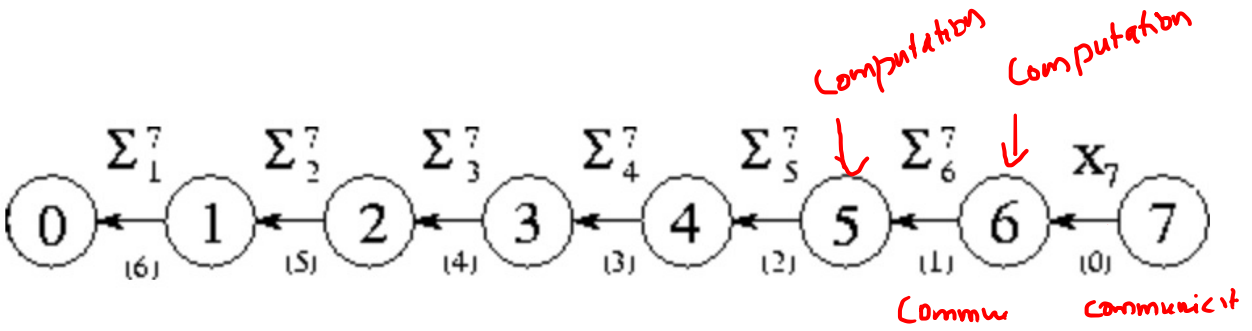
Accumulator

$S = \sum_{i=0}^{N-1} x_i$

(2) Sequential

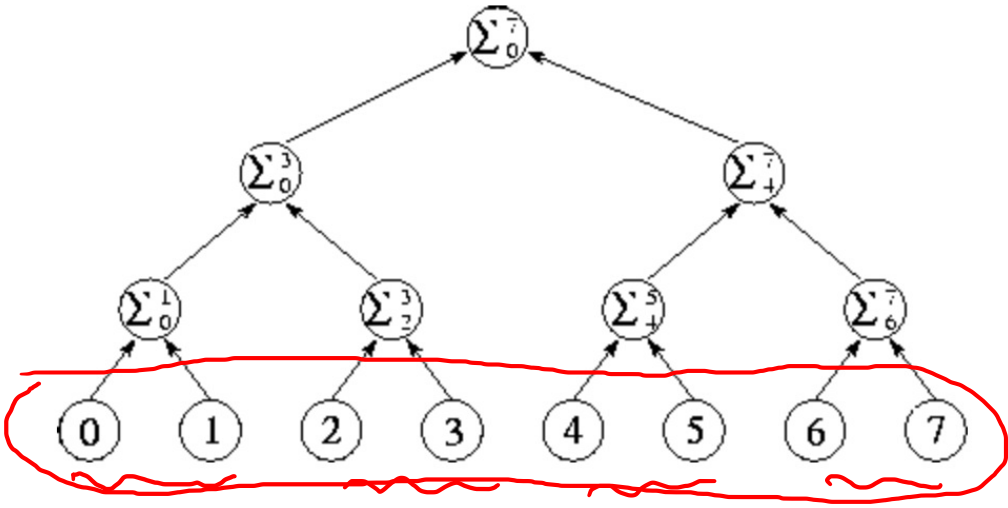


{ Concurrency }



Methodical Design of Parallel Algorithms

Divide & Conquer { log n } concurrency

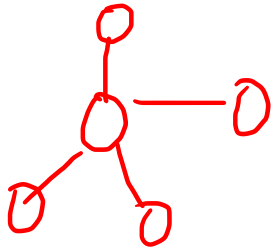



```
Procedure DAC
{
    if base case
        solve the problem
    else
        partitioning problem into L & R
        Solve L using DAC
        Solve R using DAC
        Combine L & R to get solution
}
```

Unstructured

graphs

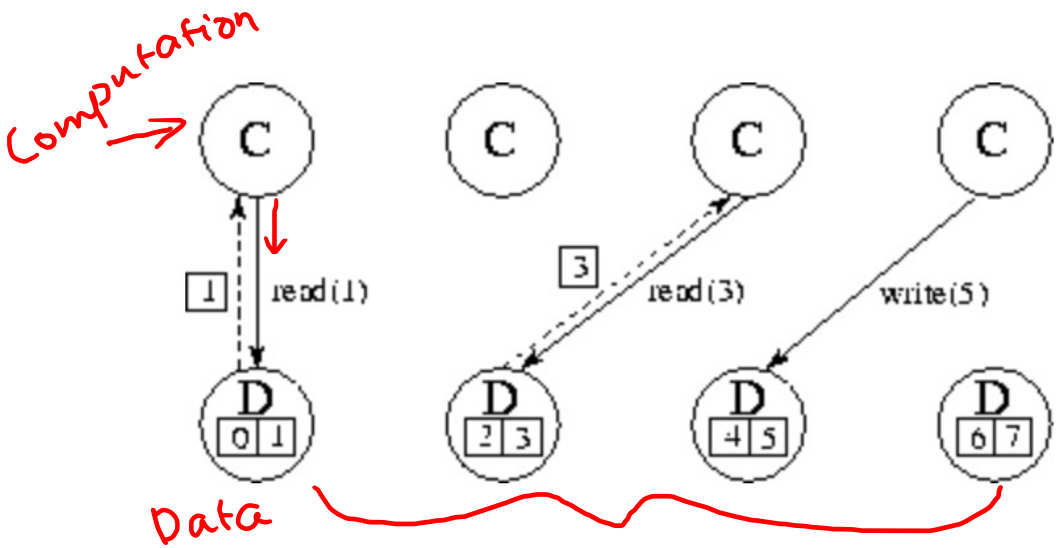
mesh



Dynamic

Methodical Design of Parallel Algorithms

Asynchronous or Synchronous.



1. polls polling (C) DS
2.

1. A[i]
2. A[i] x B[i]
3. → request C ← polling
4. B[j] x C[j]
5. :
6. request D ← polling

Communication —

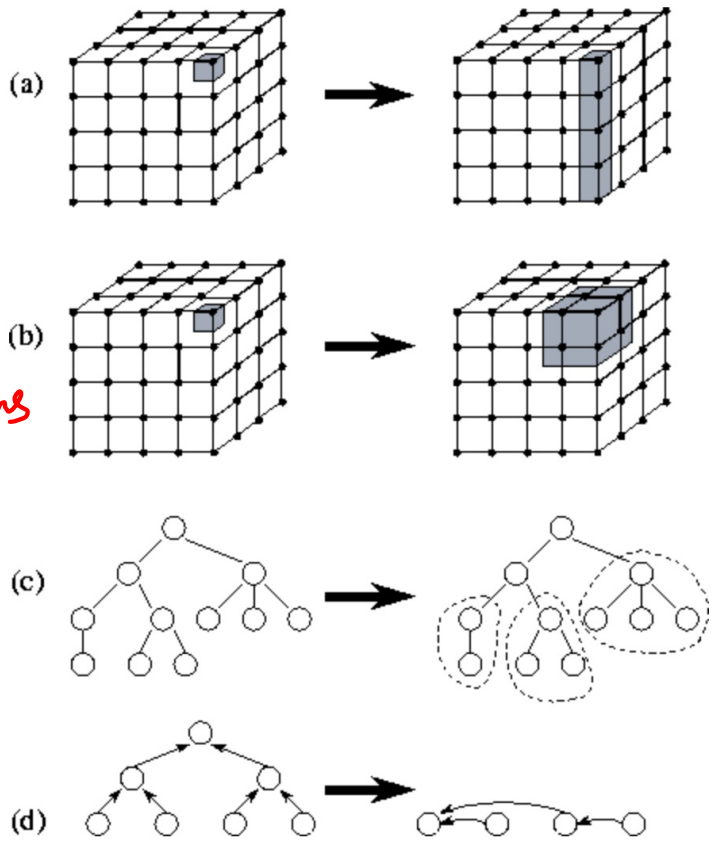
1. Unbalanced → Scalability ?
2. global → local
3. efficiency → Concurrency
   ↘ PAC

Methodical Design of Parallel Algorithms — Agglomeration

Determine → replicate the data

1. reduction in communication cost
   increasing computation
2. retain the flexibility wrt
   scalability and mapping decisions
3. Reduce the S/w Engineering Cost

task < processors



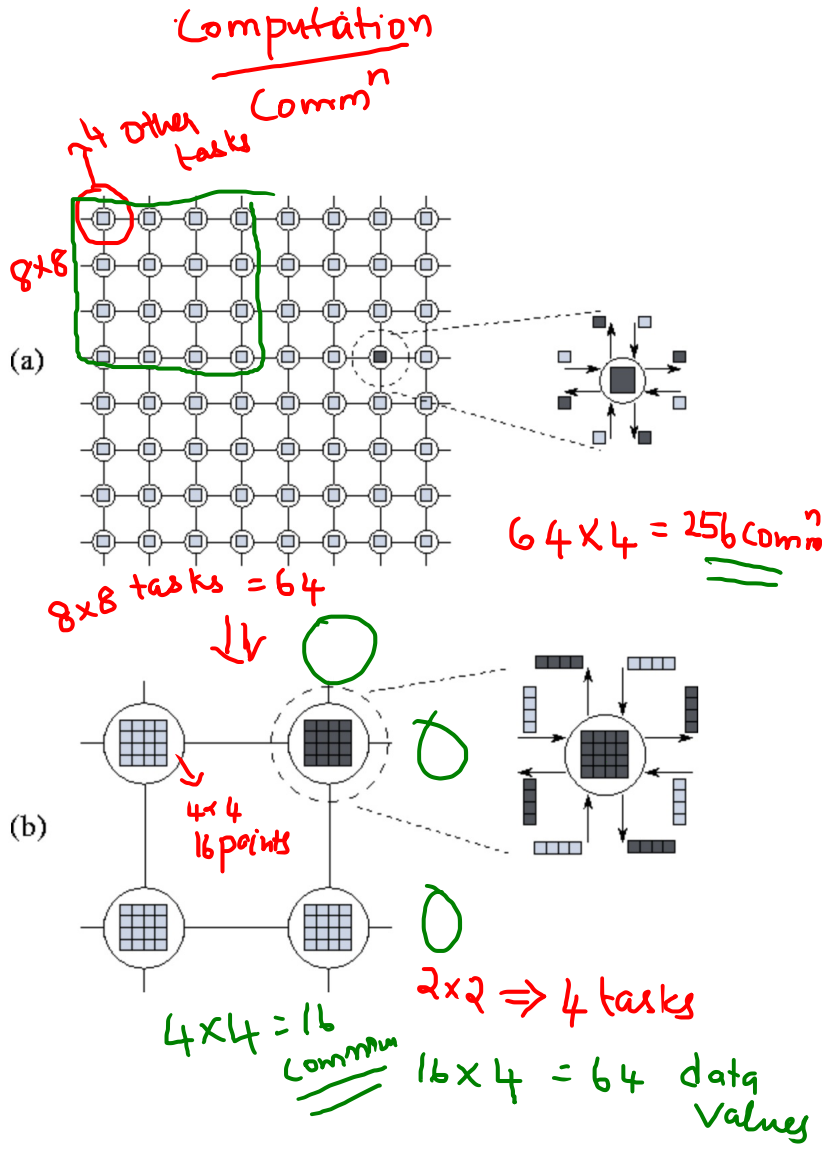
Methodical Design of Parallel Algorithms

1. Increasing Granularity

Surface-to-volume effects

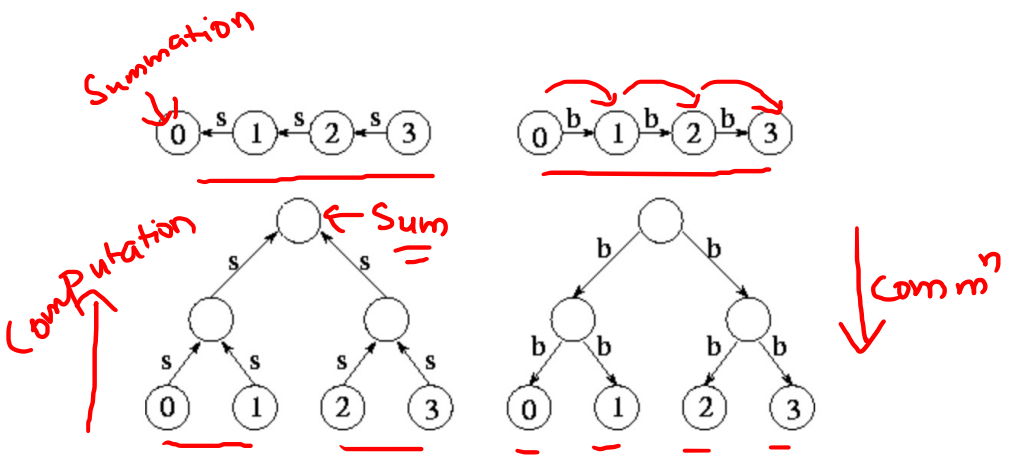
Replicating Computations

0 1 2



Methodical Design of Parallel Algorithms

Sum → replicate



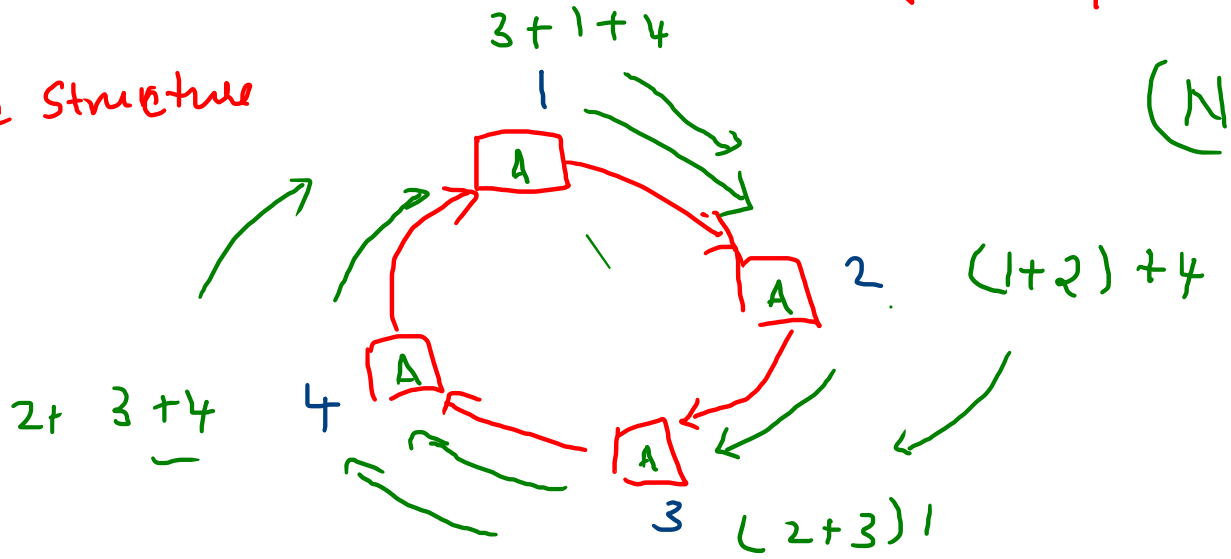
S - Summation

b - broad casting

- 2(N-1) steps

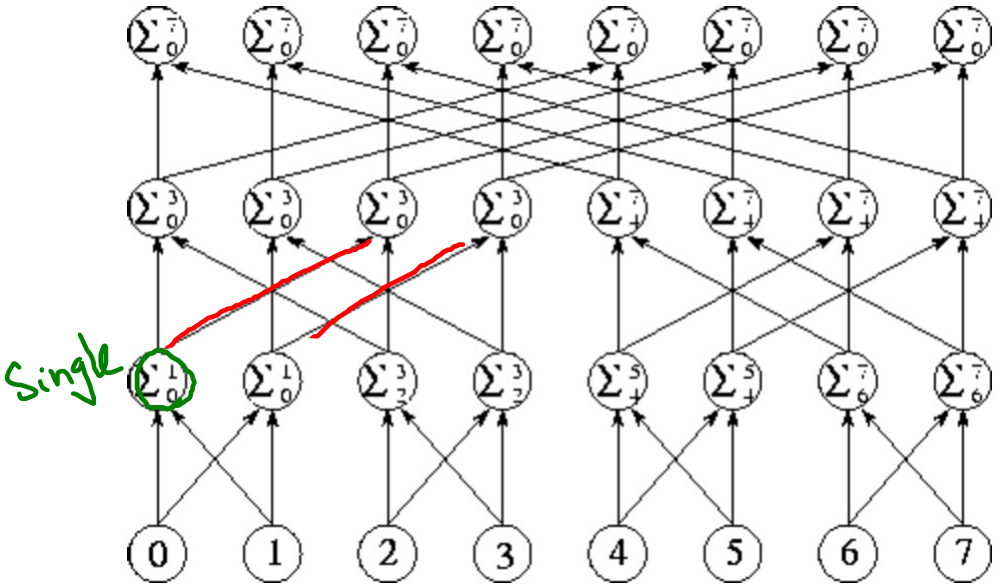
- 2 log N steps.

ring based structure



(N-1)

Butterfly Comm^n



'n' tasks ⟹ Single processor

more processors

↙ waiting for remote data

↘ { overlapping comp^n & comm^n }

2 // more no: of tasks than processors

{ mapping strategies }

↳ load balance

4 - Mapping

mapping algms → minimize the total execution time

Goal

(1) To execute concurrently on different processors.
(2) Tasks which communicate frequently on the same processor, which increases locality.

Load balancing algms

Dynamic load balancing strategy - Agglomeration & mapping

Probabilistic load balancing

{local algorithms}

{Global computation state}

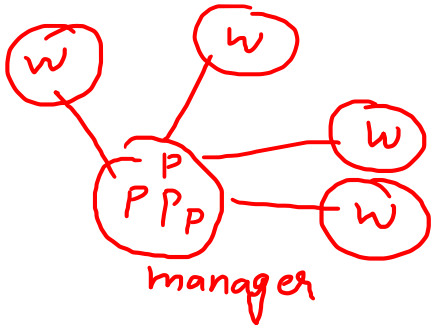
{domain decomposition}

master/slave

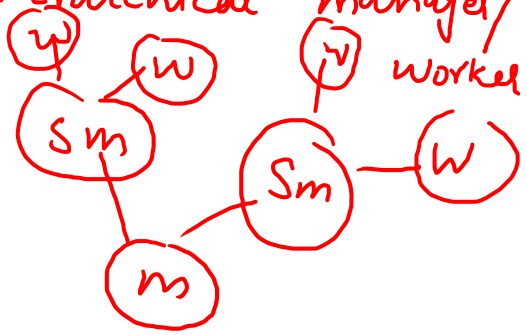
Functional decomposition } → task-scheduling Algm →

Decentralized

RR



hierarchical manager/worker



RAM model

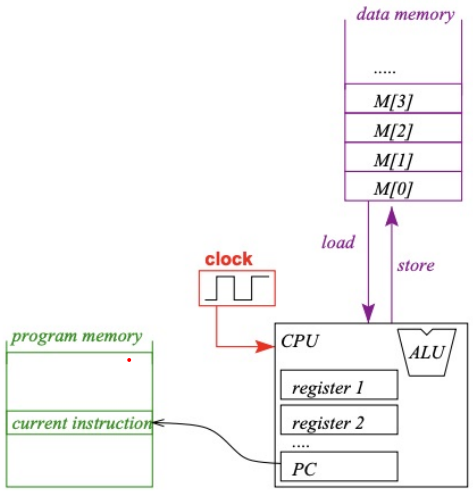
Count the primitive operations

*

frequency count

=

$O(n)$



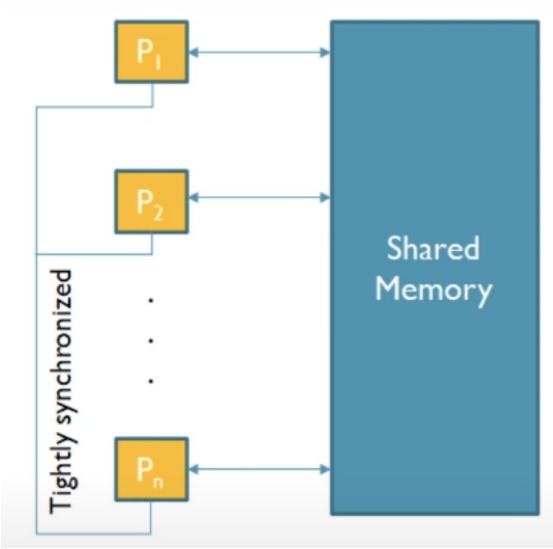
s = sum

$O(n)$

```
s = d(0)                    ——— C
do i = 1, N-1               ———
   s = s + d(i)             ———
end do                      ———
```

PRAM

1. P processors connected to single shared m/y
2. processor id $(1 \le i \le p)$
3. SIMD



multiprocessor system with shared memory, on which all n processors operate in lockstep in memory access and program execution operations. Each processor can access any memory location in unit time

PRAM Model Variants

Writes \ Reads	Exclusive	Concurrent
Exclusive	**EREW** Least effective Most simple	**CREW** Most common
Concurrent	**ERCW** Practically no use	**CRCW** Most powerful Most complex

1. EREW → Exclusive Read (Upper bound)
   - Exclusive Write
   - does not allow any kind of simultaneous to a single m/y location

2. CREW → Concurrent Read exclusive Write

3. ERCW → Exclusive Read Concurrent Write.

4. CRCW — Concurrent Read & Write. (Lower bound)

CRCW → How to resolve concurrent writes ?

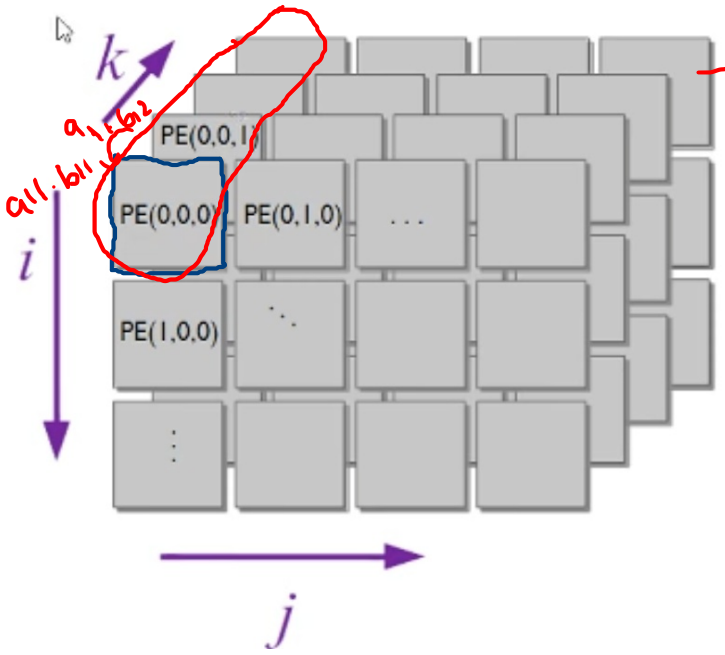
↳ (1) Common CRCW → Concurrent writes are allowed when processors are attempting to write the same value

(2) Arbitrary CRCW → a value arbitrarily chosen from the values written to the common m/y location

(3) priority CRCW →

(4) Combining CRCW →

Matrix multiplication — PRAM



$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

→ individual multiplication

↓

$O(n^3)$

↓

Strassen's multⁿ

↓

$O(n^{2.81})$

SIMD

$\{ C_{ij} = A_{ik} \times B_{kj} \}$

↓

$O(1)$

Individual processors

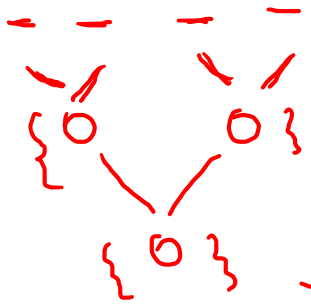
Step 1

1. Read $A(i, k)$
2. Read $B(k, j)$
3. Compute $A(i, k) \times B(k, j)$
4. Store in $C(i, j, k)$

Step 2

1. $\ell \leftarrow n$
2. Repeat
   $\ell \leftarrow \ell/2$
   if $(k < \ell)$ then
   begin
   Read $C(i, j, k)$
   Read $C(i, j, k + \ell)$
   Compute $C(i, j, k) + C(i, j, k + \ell)$
   Store in $C(i, j, k)$
   end
   until $(\ell = 1)$

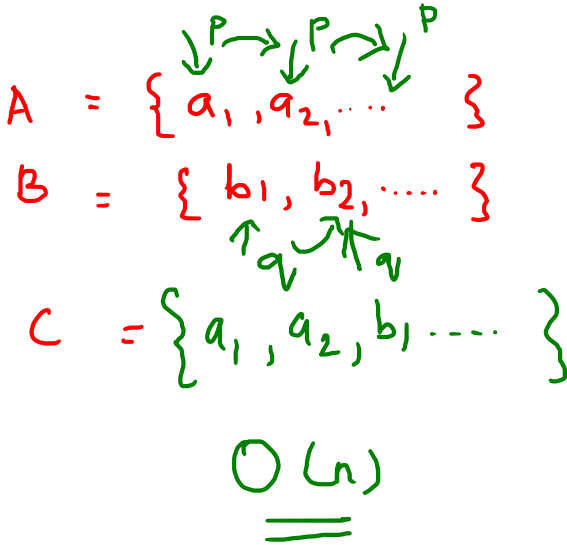
Summation of n numbers ⇓ DAC



→ $O(\log n)$

$O(n^3)$

Merging 2 sorted list



Parallel Merge

L1 L2

n/2 n/2 → n elements

O(n)

h → processors

binary Search

A[1] P_3 A[8]

A = 1, 5, [7], 13, 17, 19, 23 → Lower

B = 2, [4], 8, 11, 12, 21, 24 → Upper

A[9] A[16]

C = ?

- Processors
One for each element

Task of P_3

$(i-1) + (high - n/2)$ => position of 7 in merged list.

Binary Search : A[i = 3]

$i - 1 = (3 - 1) = 2$ elements in the lower array

7 is larger 2 elements

A[3]

high = index of the largest integer smaller than 7

index = 10 + $(high - n/2) = 10 - 8 = 2$ elements in upper array

A[1] P_3 A[8] - Processors

A = 1, 5, 7, 13, 17, 19, 23 → Lower — One for each element

B = 2, 4, 8, 11, 12, 21, 24 → Upper

A[9] A[16]

P_{11}

C = ?

Task of P_{11}

$(i-(n/2+1) + high)$

A[i=11] = 8

$i-(n/2+1) = 11-(8+1) = 2$

Binary Search with A[11] in the lower array

high = index of largest integer smaller than 8

high = 3

A[1] P_3 A[8]

A = 1, 5, 7, 13, 17, 19, 23 → Lower

- Processors
One for each element

B = 2, 4, 8, 11, 12, 21, 24 → Upper

A[9] A[16]

P_{11}

C = ?

Task of P_{11}

$(i-(n/2+1) + high)$

A[i=11] = 8

$i-(n/2+1) = 11-(8+1) = 2$

Binary Search with A[11] in the lower array

high = index of largest integer smaller than 8

high = 3

Array C

```
MERGE.LISTS (CREW PRAM):

Given: Two sorted lists of n/2 elements each, stored in
       A[1] ··· A[n/2] and A[(n/2)+1] ··· A[n]
       The two lists and their unions have disjoint values
Final condition: Merged list in locations A[1] ··· A[n]
Global  A[1 ··· n]
Local   x, low, high, index
begin
  spawn (P1, P2, ..., Pn)
  for all Pi where 1 ≤ i ≤ n do
    { Each processor sets bounds for binary search }
    if i ≤ n/2 then
      low ← (n/2) + 1
      high ← n
    else
      low ← 1
      high ← n/2
    endif
```

Binary search

```
    { Each processor performs binary search }
    x ← A[i]
    repeat
      index ← ⌊(low + high)/2⌋
      if x < A[index] then
        high ← index − 1
      else
        low ← index + 1
      endif
    until low > high
    { Put value in correct position on merged list }
    A[high + i − n/2] ← x
  endfor
end
```

? T/F

{ lower array }
{ upper array }

Parallel Merge

Time Complexity ?

(const + BS) → one element

Sequential algm

O(n)

BinaryS = O(log n)

Overall TC = O(n log n)

```
MERGE.LISTS (CREW PRAM):

Given: Two sorted lists of n/2 elements each, stored in
       A[1] ··· A[n/2] and A[(n/2)+1] ··· A[n]
       The two lists and their unions have disjoint values
Final condition: Merged list in locations A[1] ··· A[n]
Global  A[1 ··· n]
Local   x, low, high, index
begin
  spawn (P1, P2, ..., Pn)
  for all Pi where 1 ≤ i ≤ n do
    { Each processor sets bounds for binary search }
    if i ≤ n/2 then
      low ← (n/2) + 1
      high ← n
    else
      low ← 1
      high ← n/2
    endif
```

```
    { Each processor performs binary search }
    x ← A[i]
    repeat
      index ← ⌊(low + high)/2⌋
      if x < A[index] then
        high ← index − 1
      else
        low ← index + 1
      endif
    until low > high
    { Put value in correct position on merged list }
    A[high + i − n/2] ← x
  endfor
end
```

Array C

Binary search

? T/F

{lower array}
{upper array}

Parallel Merge

Time Complexity ?

(const + BS) → one element

Sequential algm

O(n)

BinaryS = O(log n)

Overall TC = O(n log n)

Work - Depth Model → DAG

↳ Topological ordering

Single processor

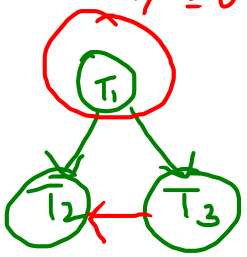
Time taken ∝ number of operations ← work

T_1 → amount of time algm take on one processor

T_p → " p processors.

↳ wait for the slowest processor to finish

This is known as the depth of an algm



→ Parents visited before the child node

→ Indegree = 0

→ prerequisite ↓ Course

Work :- no: of computations x no: of processors used

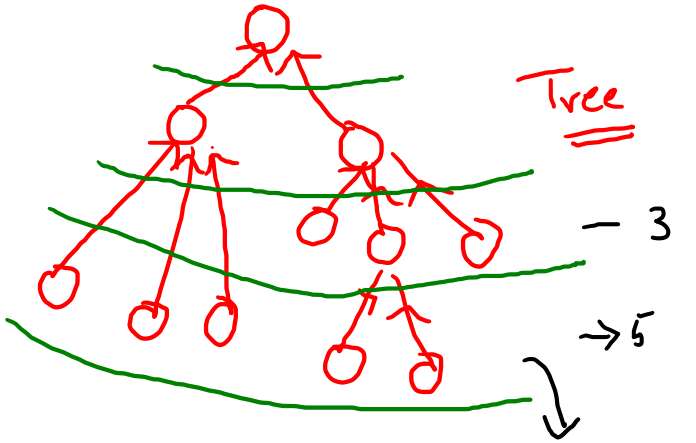
Lower bound

$$\frac{T_1}{p} \leq T_p$$

Depth of PRAM with Infinite Processors

- Representing algms as DAGs
  - multiple computations Vs Infinite processors
  - representing the dependencies b/w operations in an algm using a DAG.

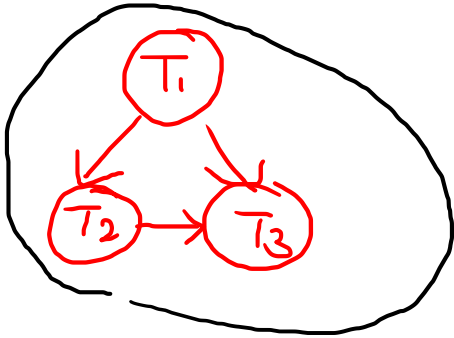
Constructing a DAG from an algm.



Topolo-

DAG

Concurrently ?



DAG

depth = T_∞ = depth of computation DAG.