myNotes
27/Aug

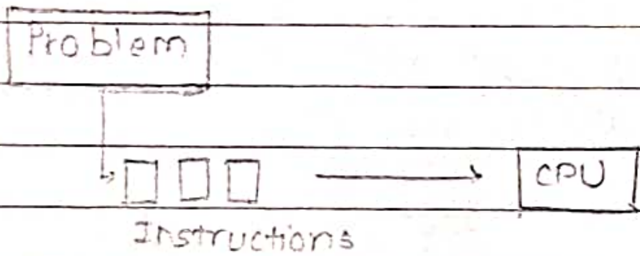
2. Principles of Parallel Algorithm Design

Parallel Algorithm Design

-
-
-
-
-

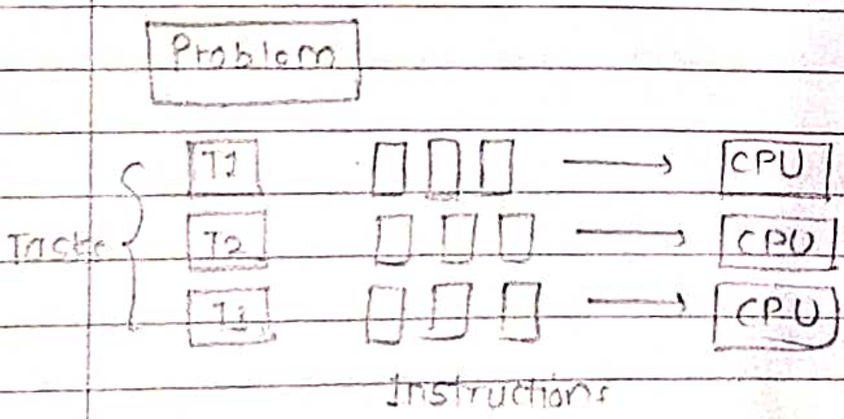
Serial Computing:

- Runs on single comp. with single CPU.
- Problem broken into set of instructions.
- Instructions are executed one after another.



Parallel Computing:

- Runs using multiple CPUs.
- Problem is broken into parts
- Each part further broken into instructions.



- uses parallel comp. architectures = Clusters, GPU's etc.

Why Parallel Computing :

- saves time
- Provides concurrency
- solves larger problems . .

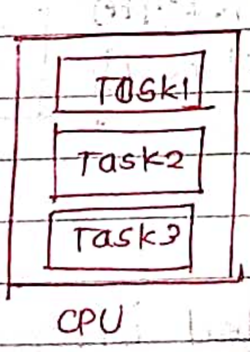
Parallel Processing :

- Technique used in parallel computing.
- General technique where task is divided into smaller sub-tasks which can be executed concurrently.
- can be applied in single computers, Multicore processors, distributed comp. environments.

Parallelism vs Concurrency :

• Concurrency :

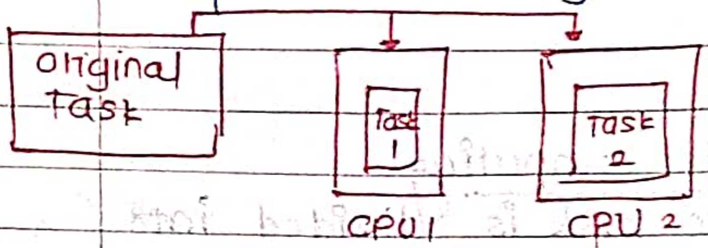
- Allows multiple tasks / Processes to be executed concurrently on single CPU.
- These tasks are managed by OS.
- Rather than completing one task entirely before moving to the next, CPU switches betn tasks at a very fast rate. (it seems tasks are running in parallel manner)



• Parallelism

- Breaking down a larger task into smaller, independent sub-task that can be processed concurrently by multiple processors or cores

- It enables the system to perform multiple operation "truly" simultaneously.



▸ Parallelism :

1] Data Parallelism

- Processing different parts of same data simultaneously.
- Useful where same operation is need to apply on multiple data elements concurrently.

Example

sum of ⓑ array and ⓒ array into ⓐ array

a[30], b[30], c[30]

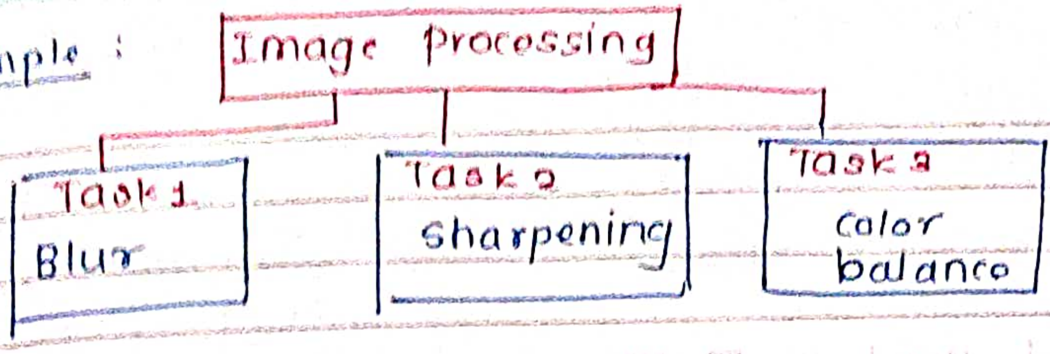
data dividing

Task 1	Task 2	Task 3
for (i=0; i< 10; i++) a[i] = b[i] + c[i]	for (i=10; i<20; i++) a[i] = b[i] + c[i]	for (i=20; i<30; i++) a[i] = b[i] + c[i]

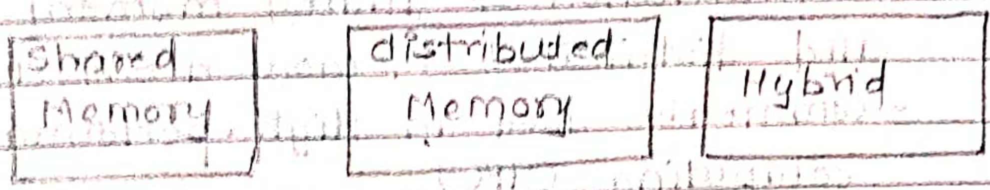
2] Task Parallelism :

- larger task divided into Smaller sub-tasks.
- Useful where independent tasks need to perform simultaneously.

Example :



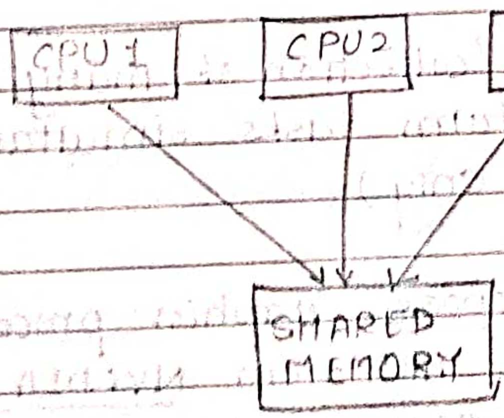
- Parallel Processing in Memory / Parallel Processing architectures :



① SHARED MEMORY :

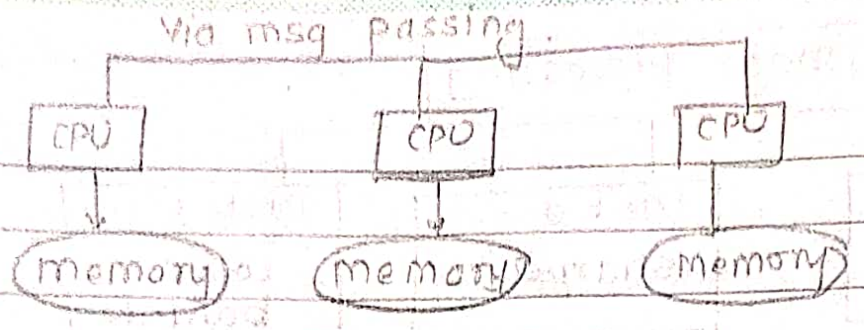
- Multiple processors shares common memory space.
- communication : By reading from or writing to shared memory area.

- Programming model = Multithreading.



② Distributed Memory :

- Multiple processors operate independently, where each having its own separate memory space.
- Communication = MPI (Message Passing Interface)



Model used - Multicore

③ Hybrid Architecture :

- Combination of shared memory and distributed memory architecture.
- Commonly used in High-performance computing (HPC)

keywords : GPU, GPGPU, NVIDIA

GPU

- Graphics Processing Unit
- Handles graphics related tasks
- NVIDIA well-known manufacturer of GPU's.
- GPU is team / collection of many cores that can perform tasks simultaneously. (Graphics related tasks only)

GPGPU

- General purpose Graphics processing Unit.
- It is power GPU from NVIDIA
- It perform many tasks more than just graphics related tasks.
- GPGPU : Collection of many cores that can perform many tasks (not only Graphical tasks)

- Processing Unit Speed :
  - Processing speed of computer is measured in FLOPS
  - FLOPS - (Floating-Point Operations Per Second)
  - FLOPS depends on
    - H/w
    - s/w
    - Specific instruction.
  - Formats:
    - FP16 (Half Precision)
    - FP32 (Single Precision) — mostly used.
    - FP64 (Double Precision)

- Parallel System Performance :

1] Speed Up : Measures how much faster a parallel system can solve a problem compared to a sequential system

- Speed Up = Execution time of single / Execution time for P
- $S(N,P) = T(N,1) / T(N,P)$
  $= T_s / T_p$

Here P = No. of processors
N = Size of problem
$S(N,p)$ = Speed up for N size problem with P processors.
$T(N,1)$ = Execution time taken by 1 processor
$T(N,P)$ = Execution time taken by P processors

- Ideal : $1 <= S(N,P) < P$

2] Parallel Efficiency :

- How effectively the processors in a parallel system are utilized
- Parallel Efficiency = Speed Up / No. of processors

$$E(N,P) = S(N,P) / P$$

- $\boxed{E(N,P) \leq 1}$

3] Scaling ~~on~~ -

- Describes how runtime of a parallel application changes as the no. of processors increased.

- strong scaling
  - Problem size (N) remains constant
  - No. of processor increased (P) $P\uparrow$
- Weak scaling
  - Increase in Problem size (N) $N\uparrow$
  - Increase in Processors (P) $P\uparrow$

▶ Amdahl's law :

- In order to understand limits of performance improvement when making a program faster by parallelizing it.

- Amdahl's law tell us that no matter how many processor we add to speed up, the overall speed will always limited by the portion of program which must need to done in sequentially (one after another)

i.e some tasks are needed to done in sequentially fashion only.

Page No.: Date: YOUVA

* speed Up (s) $= \dfrac{1}{(1-p) + \left(\frac{p}{N}\right)}$

- Here p is fraction of program that can be parallelized (if 70% then $p = 0.7$)
- N - no. of processors.

* speed up (s) $= \dfrac{1}{(a) + \left(\frac{(1-a)}{N}\right)}$

- Here a is fraction of program that can't be parallelized

Ex Find Speed Up (s)
if $a = 0.1$
$N = 16$ (No. of processors)
$S = ?$

① $S = \dfrac{1}{a + \left(\frac{(1-a)}{N}\right)} = \dfrac{1}{0.1 + \frac{0.9}{16}} = 6.4$

if $N = 1024$

$S = \dfrac{1}{0.1 + \frac{0.9}{1024}} = 9.91 \cong 10$

Ex $N = 4, 16, 1024$ (No. of processors)
$a = 0.2$
find $S = ?$

i] $S = \dfrac{1}{0.2 + \dfrac{(1-0.2)}{4}}$

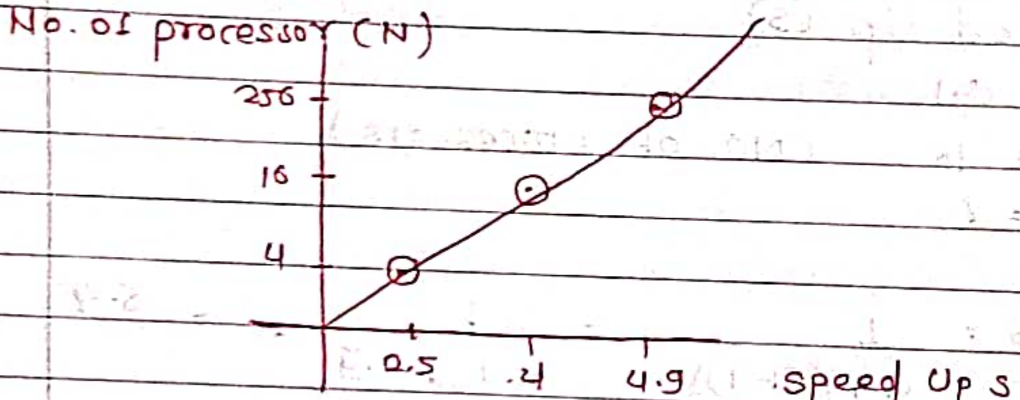
$= 2.5$ — for $N = 4$

ii] $S = \dfrac{1}{0.2 + \dfrac{(1-0.2)}{16}}$

$= 4$ — for $N = 16$

iii] $S = \dfrac{1}{0.2 + \dfrac{(1-0.2)}{256}}$

$= 4.9$ — for $N = 256$



* Flynn taxonomy : classification of computer architecture on the basis of no of instructions and data.

① SISD - Single instruction, single data
② SIMD - Single instruction multiple data
③ MIMD - Multiple instructions, multiple data
④ MISD - Multiple instructions, single data

Work Span Model :

- Used for analyzing performance & parallelizability of algorithm.
- Helps to understand how well an algo can divided into smaller tasks that can be executed concurrently on multiple processing units.

Total Work = T_1 : Amount of work required to complete the task when using a single processor

Span (Critical path) T_∞ : Determines minimum time required to complete the task, considering longest sequence of dependent tasks.

▸ Work law :

$$\frac{\text{total work } (W)}{\text{No. of processors } (P)} >= \text{time taken by one processor to complete the task } (T_1/P)$$

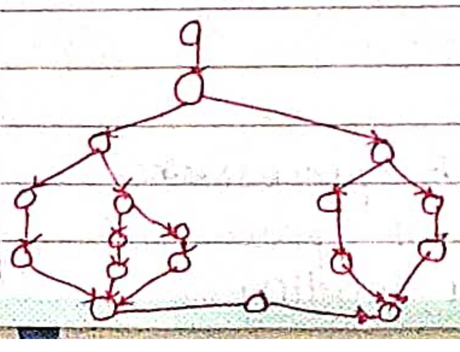
- $W/P >= T_1/P$

or

- $\boxed{T_P >= T_1/P}$

▸ Span law : No matter how many processors you used, the execution time (T_P) cannot less than the span (T_∞)

- $\boxed{T_P >= T_\infty}$



Parallel Programming Using MPI:

- MPI (Message Passing Interface)
  - Used as standard for parallel programming.
  - Library that enables processes in a distributed computing environment to communicate & synchronized with each other.

Steps :

1] Installation of MPI on your system.

2] Write code and save using .c or .cpp

3] compile :

mpicc -o program program.cpp

4] run

mpiexec -n <no.of processes> ./program

Structure

```
# include <mpi.h>          — Header file
MPI_Init (&argc, &argv)     — Initialization
MPI code
  — " —
  — " —
MPI_Finalize()              — End of code.
```

Some MPI functions :

MPI_Init()
MPI_Finalize()
MPI_Comm_Size() — No. of processors
MPI_Comm_rank() — rank of processor
MPI_Send () — data sending.

MPI_Recv() —— Receiving the data

MPI_Barrier() —— Synchronizes all processors.

▸ **Parallel Sum : (Considering master-slave parallel computing model)**

- No. of Elements a1, a2 ____ an are given.
- Compute the sum using multiple processors.
- In master slave scenario,
  - master process divides array into smaller segments & assigns each segment to a slave process.
  - Each slave process computes the sum of its assigned segment
  - Master then combines the results from all slaves to get final sum.

$\boxed{n/P}$

n = elements

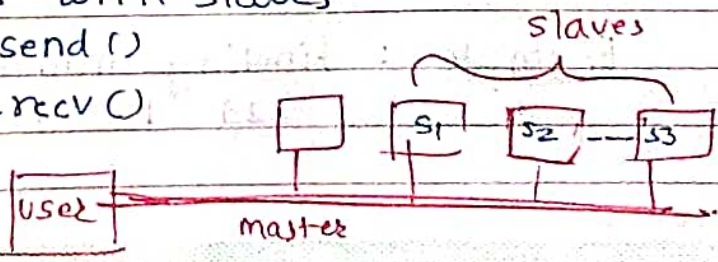
p = processor

(elements per processor)

Master (rank=0)
- slave 1 (rank=1)
- slave 2 (rank=2)
- |
- slave p (rank=p)

if (rank==0)
master
else
slave

master communicates with slaves using
① MPI_send()
② MPI_recv()



▶ Granularity :
- Decides the size of individual tasks within parallel application.

Granularity

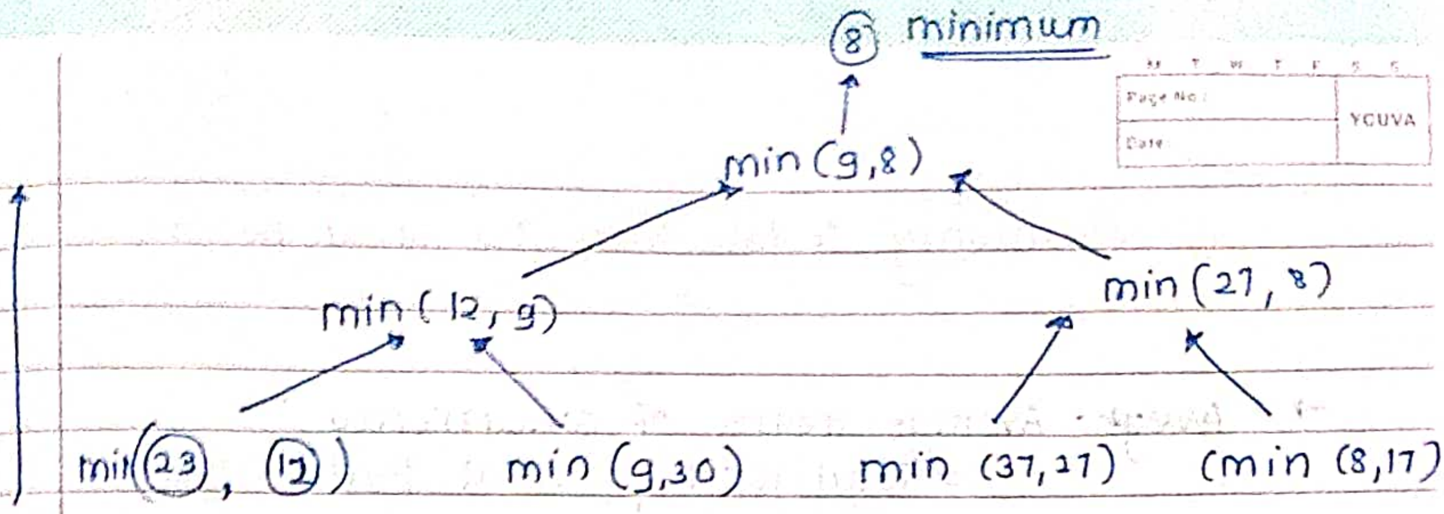
Finer Grained Granularity
- tasks are divided into very small units.
- overhead in communication & synchronization

Coarse-Grained Granularity
- tasks are divided large and more significant units.
- reduces the overhead.

▶ Task Dependency Graph :
- Decomposition
- Tasks

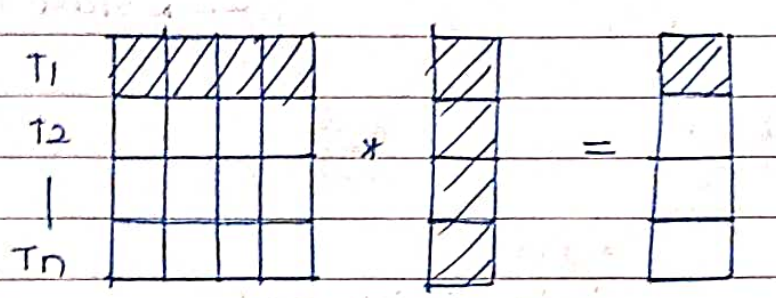
- First step for developing parallel algorithm is to divide the problem into tasks.
  or decompose
- Those decomposed parts called Tasks
- Task Dependency graph :
  Directed acyclic graph where
  - Nodes - are tasks
  - edges - are dependencies.

Example : Finding minimum among
23 12 9 30 37 27 8 17

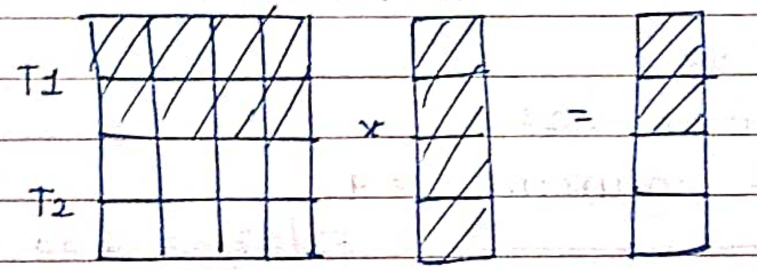


→ Granularity : Example (Dense matrix multiplication with a vector)

Number and size of the tasks into which a problem is decomposed



Fine-Grained Granularity



Coarse -Grained Granularity

→ Concurrency :

- Executing multiple tasks simultaneously, called concurrency.

→ Maximum Degree of concurrency.

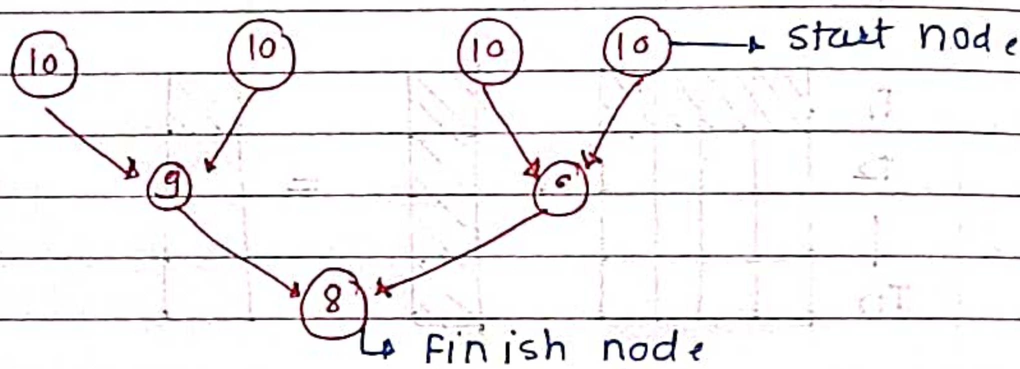
→ max. no. of tasks that can be executed simultaneously in parallel program at any given time is known as max degree of concurrency.

→ due to dependencies among tasks max. degree of concurrency is less than the total no. of tasks.

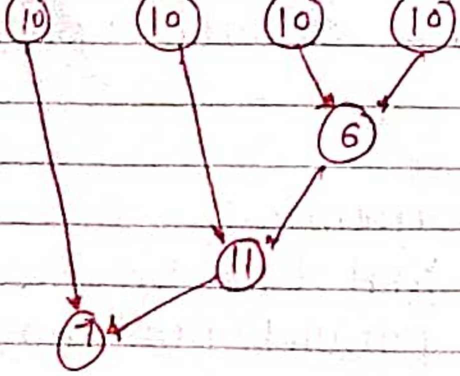
→ ~~Avagr~~ · Average degree of concurrency :
= Total work / critical path length.

1] Critical Path : distance betⁿ Start 4 finish node with max weight

2] Critical Path Length : sum of weights of critical path.



1) Total Work = 63
2) Critical Path Length = 27
3) Max degree of concurrency = 4
4) Avg —11— = 63/27 = 2.33 = 2
5) Min processors = 4



1] Total work = 64
2] = 34
3] = 4
4] = 64/34 = 1.88
5] = 4