UNIT IV HPC

Analytical modeling of parallel Programs

Sources of Overhead in Parallel Programming -

different types of overheads -

1) Inter process communicaⁿ - In any non-trivial parallel task system, interaction b/w communicaⁿ bet processing elements are crucial for sharing data. The time spent communicating data between processing ~~overhead~~ units slows down parallel processing.

2) Idling - In parallel systems, processing elements can sit idle for various reasons like unequal workloads, need for synchronizaⁿ or certain parts can't be parallalized. If different processing elements have different workloads some processing elements may be ~~idle~~ idle during part of the time that others are working on a problem. In some ~~parallel programs processing elements may be idle during~~ In some parallel programs processing elements must synchronize at certain points during parallel execuⁿ

3) ~~Excess computaⁿ - Parallalizing a problem may necessitate~~

3) Excess Computaⁿ - When the best sequential algorithm for task is hard to parallelize, we're left ~~to with~~ with less efficient parallel option. This means the parallel program does more work than optimal one, resulting excess computaⁿ.

Teacher's Sign.: _____

Different Performance metrics

1) Speedup - Speedup measures how much faster a parallel program runs compared to its sequential counterpart. It's calculated as the ratio of execn time of sequential prog. to the execn time of parallel prog.

$$S = \frac{Ts}{Tp}$$

2) Efficiency - measures utilizan of resources in parallel system. Ratio of the speedup achieved by parallel program to no. of processor used.

$$E = \frac{S}{\text{Procesing ele}}$$

3) Overhead - Refers to additional time or resource consumed by parallalization related activities such as Communican, Sync, & load balancing. minimize overhead to acheive optimal performance.

4) Throughput - Measures the rate at which tasks or computan are completed in parallel system over given period.

5) Resource utilizan - Measures ~~item~~ extent to which system resources, such as processors, memory, & network bandwidth are utilized. High resource utilizan indicates efficient use of resources.

Teacher's Sign.: _____

A Scalability of Parallel Systems.

Algorithm are said to be scalable if the level of parallelism increases at least linearly with size of problem.

A parallel computer system is said to be scalable if its efficiency can be fixed by simultaneously increasing machine size and the problem size.

> The Isoefficiency Metric of Scalability -

1) For a given problem size, as we increase the number of processing elements, the overall efficiency of parallel system goes down

2) The efficiency of a parallel systems increases if the problem size is increased while keeping the number of processing elements constant.

Isoefficiency Function -

Parallel execution time can be expressed as a func^n of problem size overhead function, and the number of processing elements. parallel runtime.

$$T_p = \frac{W + T_o(W,P)}{P}$$

$$\text{Speedup} = S = \frac{W}{T_p} = \frac{Wp}{W + T_o(W,P)}$$

$$\text{Efficiency } E = \frac{S}{P} = \frac{W}{W + T_o(W)} = \frac{1}{1 + T_o(W,P)/W}$$

A large isoefficiency function indicates a poorly Scalable system.

Teacher's Sign.: _____

Minimum Execu^n Time - (MET)

MET refers the to the shortest possible time required to execute a given task or problem on a parallel system.

- The eq^n to find the number of processing elements for which T_P (parallel runtime) is minimum.

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

- For example - min execu^n time for adding n number parallel runtime $T_P = \frac{n}{p} + 2\log p$

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

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

Substitute $p = n/2$ we get. $T_P^{min} = 2\log n$

The processor time product for $p = p_0$ is $\Theta(n \log n)$ which is higher then $\Theta(n)$ Hence, the parallel system is not cost-optimal for the value of p that yields minimum parallel runtime.

Minimum cost-optimal parallel time -

- Let $T_P^{cost-opt}$ be the min cost-optimal parallel time
- if the isoefficiency func^n of a parallel system is $\Theta(f(p))$ then a problem of size W can be solved cost optimaly if and only if $W = \Omega(f(p))$.
in other words for cost optimality $p = O(f^{-1}(W))$
- For cost optimal system $T_P = \Theta(W/P)$ Therefore $T_P^{cost-opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right)$

Example.

- The isoefficiency func^n $f(p)$ of this parallel sys $\Theta(p \log p)$ we have $p \leq \frac{n}{\log n}$

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

T_p^{min} and $T_p^{\text{cost-opt}}$ for adding n no. are same $\Theta(\log n)$.

Amdahl's law - Amdahl's law governs the speedup of using parallel processors on a problem, versus using only one single processer, under the assumption that the problem size remains the same when parallelized.

- based on fixed problem size
- To keep efficiency of system fixed we have to increase both the size of the problem and the no. of processors simultaneously.
- It tells that for a given size, the speedup doesn't increase linearly as the number of processer increases. In fact Speed upr tends to saturated.

$$S(n) \leq \frac{1}{\underbrace{f}_{\text{Serial part.}} + \underbrace{\frac{(1-f)}{n}}_{\text{parallel part}}}$$

f is the fraction of opera^n in compata^n performed sequentially where $0 \leq f \leq 1$. The max Speedup achievable by a parallel compator with 'n' processor is

Teacher's Sign.:

Gustafson's law -

- Instead of having fixed size problem we must assume that we have a fixed execution time. This is because in case of huge problem size, we will need to increase system size ie no. of processors
- As the number of processing elements increases the total workload can also increase proportionally leading constant execn time
- Execn time is fixed let serial execn time be 's' and the parallel execn time be 'p' for 'n' processor system. Thus total execn time is s+p
- If execn to be done on sequential sys, the execn time will be s+np Thus speed up will be.

$$S'(n) = \frac{s+np}{s+p} = \frac{s+np}{1}$$

assuming time required on parallel system is 1
ie $s+p=1$ therefor $p=1-s$

$$S'(n) = \frac{s+n(1-s)}{s+(1-s)}$$

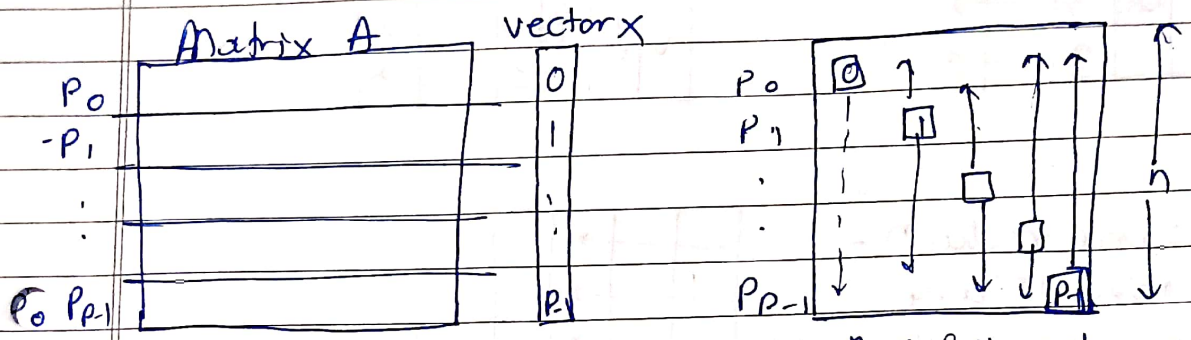
$$= \frac{n+s(1-n)}{1}$$

$$S'(n) = n+s(1-n)$$

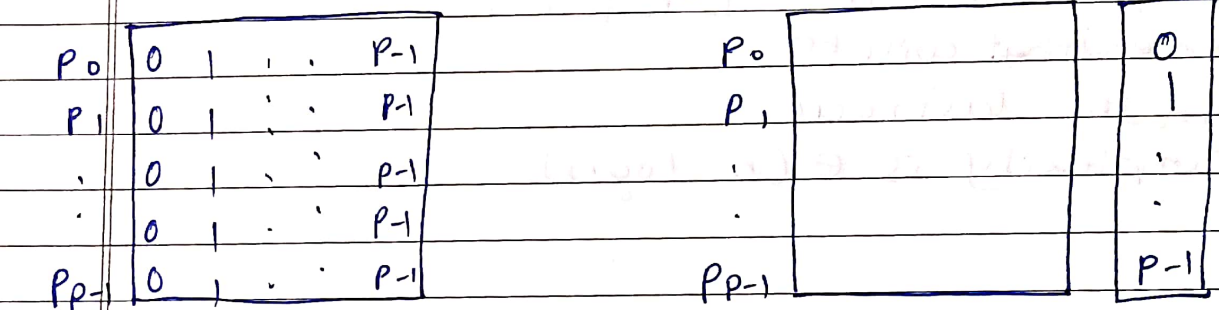
* Matrix vector multiplication -

1) Rowwise 1-D partitioning -

Required communicaⁿ - All to All broadcast.



Distribuⁿ of full vector all to all broadcast



vector distribuⁿ to each process after broadcast

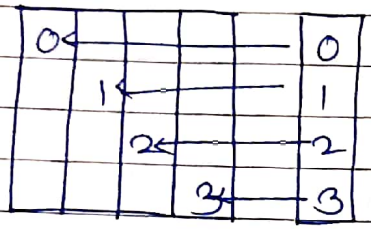
final distribuⁿ of the matrix and result vector y.

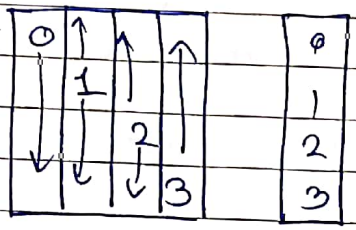
2) Rowwise 2-D partitioning

Req. communicaⁿ.

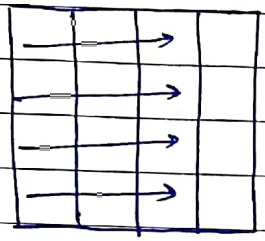
1) One to one.
2) one to All
3) All to one.

1) 1 to 1 communicaⁿ - 2) one to all broadcast.

i) 

ii) 

3) All to one reducⁿ - Whatevere results are generated in a row by a particular process in a given element that will be reduced in the last row.



Cost complexity is $\Theta(n^2 \log n)$.

Granularity - Granularity in parallel computing refers to the size or level at which a problem is divided into smaller tasks or units of work for parallel execuⁿ

effects of granularity on parallel systems.

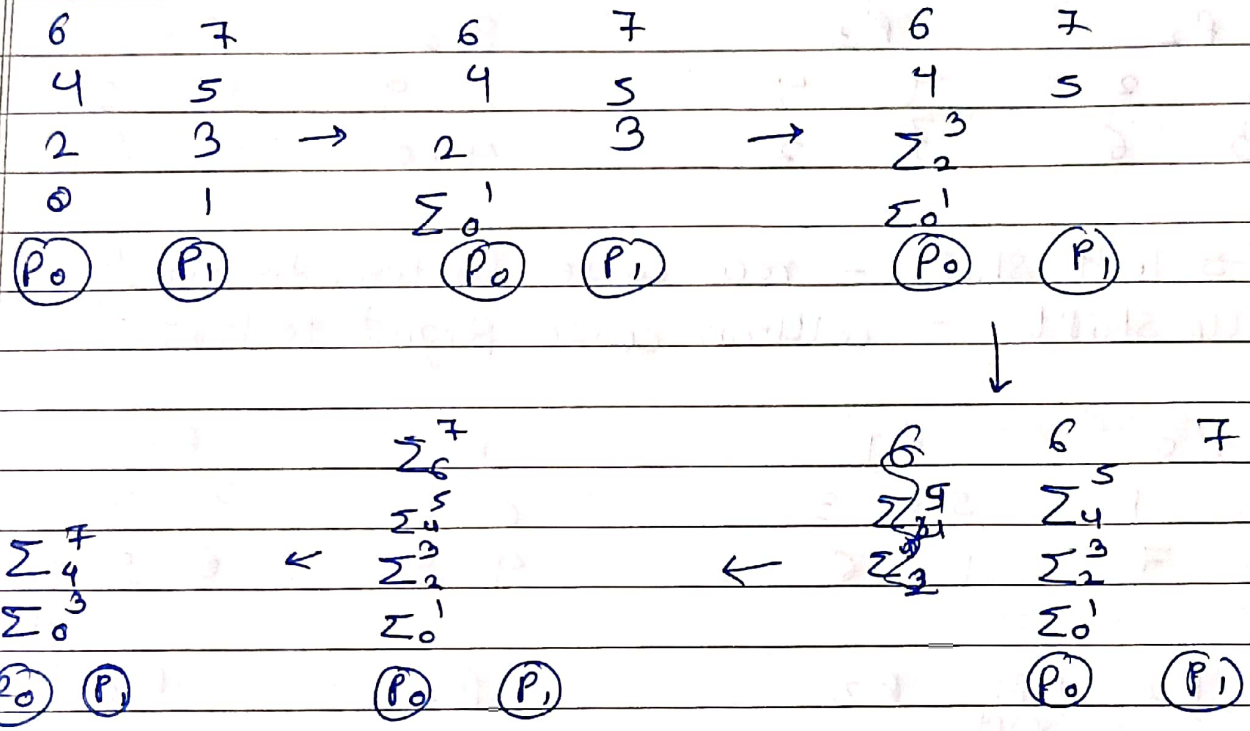
1) Task overhead - ~~It involves~~ granularity involves breaking problem into smaller tasks, which can lead to increased overhead due to task creation, scheduling & synchronization.

2) Communication overhead - granularity can result in higher communicaⁿ overhead as smaller tasks may require more frequent data exchanges & synchroniz between processing elements.

Teacher's Sign.: _____

3) Laad Balancing - Coarse-grained granularity involves larger tasks, where some processing elements have havier workloads than others. This can result underutilization of resources, reducing overall system efficiency

4) communicaⁿ decreases by Factor of n/p
~~Computaⁿ~~ ~~communicaⁿ~~ increases by factor of n/p.



P0 & P1 are physical processing element. there are 8 ~~pre~~ virtual processing units. the computation will increase by n/p ie 8/2 = 4 ie one physical processor will handle 4 processing elements.

Computaⁿ cost is $\Theta(n \log P)$ if it is performed sequentially cost will be $\Theta(n)$

ie $\Theta(n) < \Theta(n \log P)$.

Teacher's Sign.: _____

Matrix Matrix multiplicon -

$$A = \begin{bmatrix} 2 & 1 & 5 & 3 \\ 0 & 7 & 1 & 6 \\ 9 & 2 & 4 & 4 \\ 3 & 6 & 7 & 2 \end{bmatrix} \qquad B = \begin{bmatrix} 6 & 1 & 2 & 5 \\ 4 & 5 & 6 & 5 \\ 1 & 9 & 8 & -8 \\ 4 & 0 & -8 & 5 \end{bmatrix}$$

P_0		P_1	
2	1	5	3
0	7	1	6

P_0		P_1	
6	1	2	5
4	5	6	5

P_2		P_3	
9	2	4	4
3	6	7	2

P_2		P_3	
1	9	8	-8
4	0	-8	5

left shift - row wise ~~bottom to up~~

up shift - column wise ~~Right to left.~~

P_0		P_1	
2	1	5	3
0	7	1	6

P_0		P_1	
6	1	2	5
4	5	6	5

↑ up shift.

P_2	left shift ←	P_3	
9	2	4	4
3	6	7	2

P_2		P_3	
1	9	8	-8
4	0	-8	5

P_2		P_3	
4	4	9	2
7	2	3	6

P_0		P_3	
6	1	8	-8
4	5	-8	5

P_0		P_1
2	1	
0	7	

P_2		P_3	
1	9	2	5
4	0	8	5

Teacher's Sign.: _____

Final module. -

P_0	P_1
2 1	5 3
0 7	1 6.

P_0	P_1
6 1	8 -8
4 5	-8 5

P_2	P_3
4 4	9 2
3 2	3 6

matrix A

P_2	P_3
1 9	2 5
4 0	6 5

matrix B

Multiply P_0 with P_0 P_1 with P_1 and so on.

$C_0 = \begin{matrix} 16 & 7 \\ 28 & 35 \end{matrix}$ $C_1 = \begin{matrix} 16 & -25 \\ -40 & 22 \end{matrix}$

$C_2 = \begin{matrix} 20 & 36 \\ 15 & 63 \end{matrix}$ $C_3 = \begin{matrix} 30 & 37 \\ 42 & 39 \end{matrix}$

Perform left shift up shift $\sqrt{P}$ times in our case $P=4$ then $\sqrt{P} = \sqrt{4} = 2$ perform 2 times.

Prev model.

P_0 left P_1

2 1 shift 5 3

0 7 ← 1 6 ⇒

P_0	P_1
5 3	2 1
1 6	0 7

P_2 left shift P_3

4 4 ← 9 2

3 2 3 6

P_2	P_3
9 2	4 4
3 6	7 2

} mat A

P_0 P_1

6 1 8 -8

4 5 -8 5 ⇒

P_2 P_3

9 2 5

1 0 6 5

up shift, up shift.

P_0	P_1
1 9	8 -8
4 0	-8 5

P_2	P_3
6 1	2 5
4 5	6 5.

} mat B

now multiply

Teacher's Sign.: _____

matrix B.

P0		P1	
6	1	8	-8
4	5	-8	5

P2		P3	
1	9	2	5
4	0	6	5

=>

P0		P1	
1	9	2	5
4	0	6	5

P2		P3	
6	1	8	-8
4	5	-8	5

now perform corresponding ~~m~~ processors multiplicaⁿ. ~~we get~~ & adding with Previos values-

C_0 = 33 52 53 44

C_1 = 26 -14 2 57.

C_2 = 82 55 57 96

C_3 = 30 25 82 -7.

UI

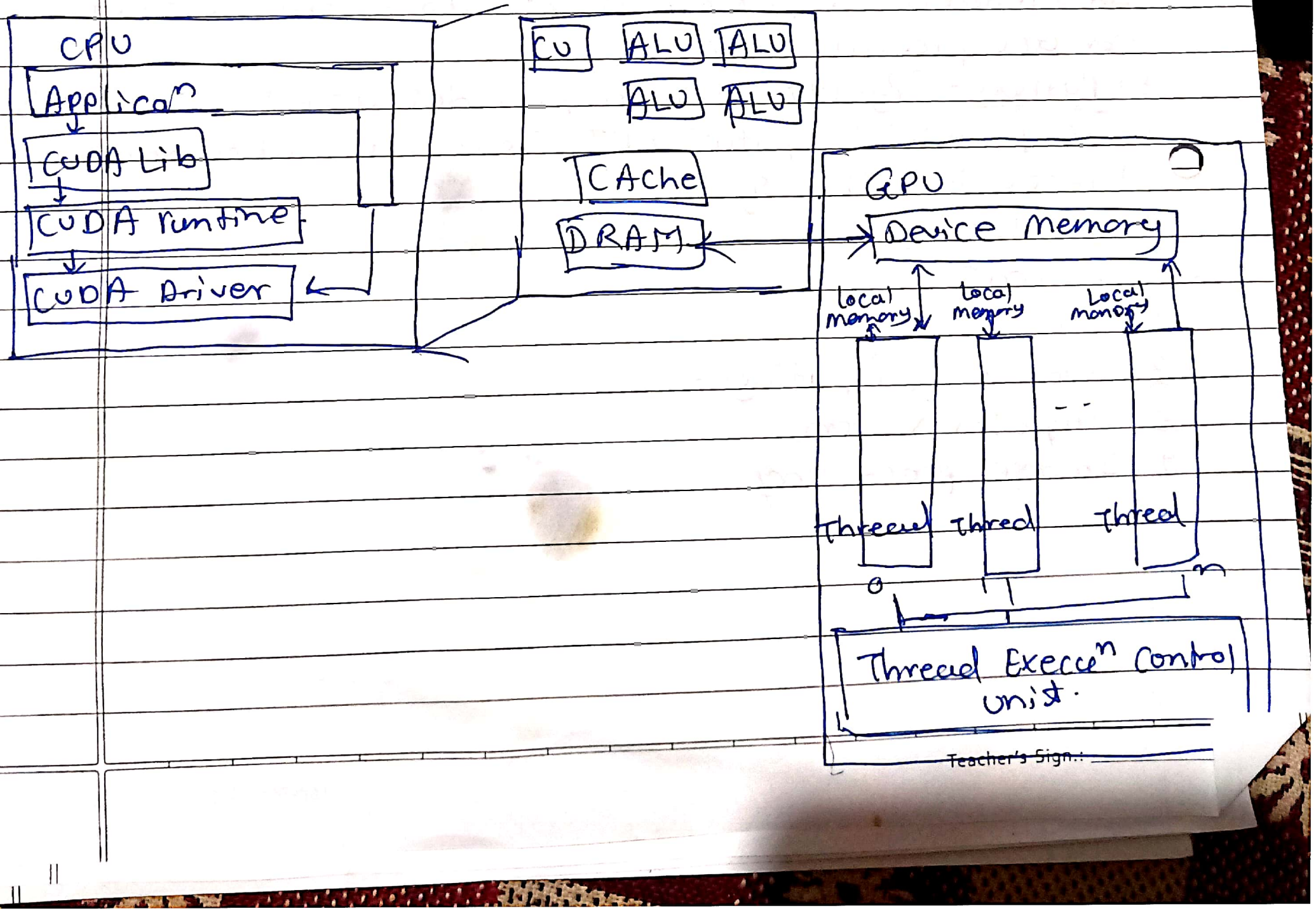
CUDA Architecture

What is CUDA?

- Compute Unified Device Architecture is a parallel computing platform and application programming interface (API) developed by NVIDIA for general computing on GPU
- CUDA provides programmers with instructions that allow GPU acceleration for data-parallel computation
- CUDA can speed up computing applications across a wide range domains including deep learning, analytics.
- Different programming langs supported by CUDA
  1) CUDA C/C++: primary lang for CUDA programming. extends C/C++ with GPU-specific features, such as kernel functions that execute on GPU & Syntax for managing memory & launching parallel threads.
  2) CUDA Fortran - CUDA Fortran compiler for ~~developers~~ scientific computing, it extends Fortran with features for GPU programming.
  3) Python - PyCUDA library provides access to NVIDIA's CUDA parallel computation API. To write CUDA kernels directly in python syntax.
  4) MATLAB

- Applications
  1) Deep learning
  2) Scientific computing -
  3) Computer vision
  4) Image processing.

Teacher's Sign.: _____


CPU
Applicaⁿ
CUDA Lib
CUDA runtime
CUDA Driver
CU
ALU
ALU
ALU
ALU
CAche
DRAM
GPU
Device memory
Local memory
Local memory
Local memory
Thread
Thread
Thread
0
n
Thread Execcⁿ Control unit.
Teacher's Sign.:

Architecture

1) CPU - Host comprises of CPU and its memory host code is executed by CPU. It allocats memory, launches kernel and handles data transfer bet^n CPU and GPU.

it contains -

1) CUDA libraries - built on top of CUDA runtime These are collections of pre-written functions & routines that provides optimized implementa?

2) CUDA Runtime - built on CUDA Driver. provides API for developers to interact with GPU

3) CUDA Driver - Low level software component responsible for communicating directly with GPU Provides interface for OS to interact with GPU.

2) GPU -

1) CPU allocates memory & prepares data for processing on the GPU. Data transferred from CPU's main memory to GPU's memory.

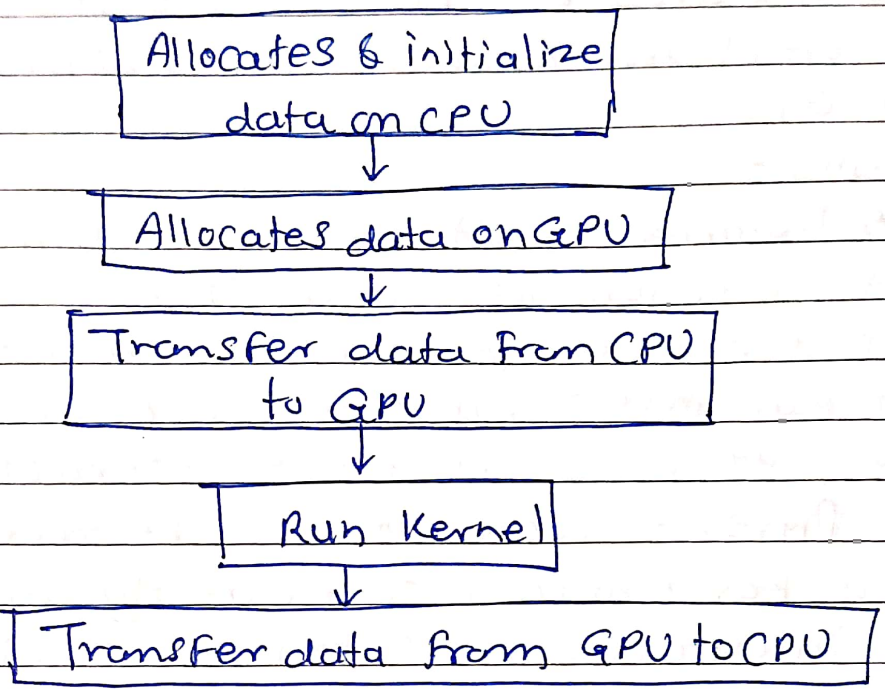
2) CPU launches CUDA kernel Func^n. Kenel launches are initiated by CPU and executed on the GPU.

3) GPU executes Kernel Function in parallel on multiple threads organized into blocks & grids.

4) After Kernel execu^n, the results are stored in GPU's memory. CPU requests the result from GPU's memory.

Teacher's Sign.:

Describe CUDA-C program Flow



1) Host (CPU) - Allocates memory on GPU transfer data betn CPU & GPU launches kernel functions and get result
2) Allocate data on GPU - CPU allocates memory on GPU for input & output data. Input transferred from CPU's main memory to GPU's memory.
3) Kernel - Kernel Function is a CUDA-C Fun^{n} that executes in parallel on the GPU. GPU launches multiple threads to execute kernel Function.
4) Retrieve result - CPU retrieves the result by transfering data back from the GPU's memory to CPU's main memory.

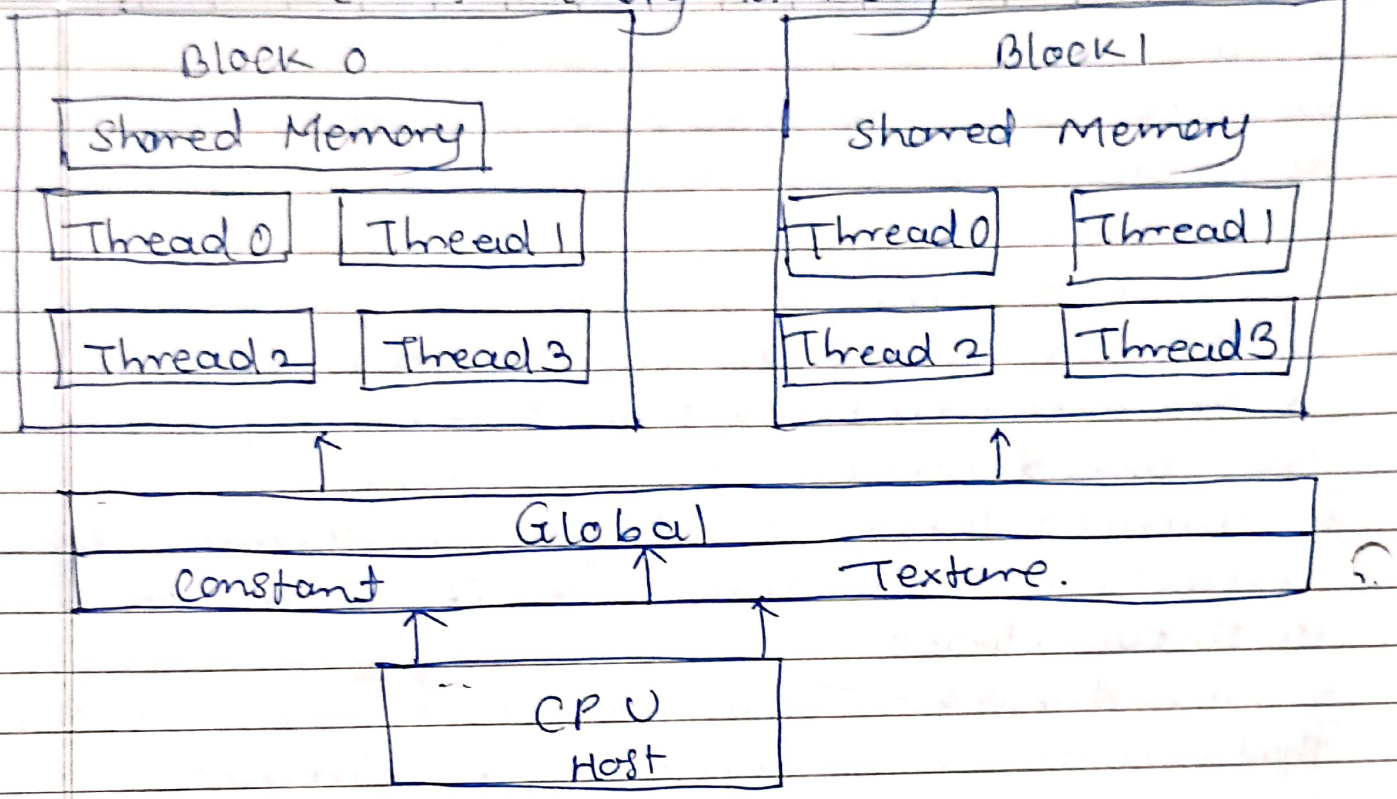
Teacher's Sign.:

Terms in CUDA

1) GPU (Graphical Processing Unit) - GPU is specialized processing device designed to accelerate the rendering of images for display devices. Used for parallel computing.
2) Kernel - Kernel is a func^n that runs in parallel on the GPU. Kernel are written in CUDA C/C++ & are executed by multiple threads in parallel. Each thread executes the same kernel
3) Thread - Thread is a single execu^n unit in GPU Threads execute in parallel on GPU & are organized in thread blocks.
4) Thread block - Thread block is group of threads that execute same kernel code. Threads within same block can communicate and synchronize using shared memory.
5) Grid - grid is collec^n of thread blocks that execute same kernel func^n. represents the highest level of parallelism in CUDA, organizing multiple thread blocks per execu^n.
6) Device - It refers to GPU used for parallel computa^n. CUDA programs can run on multiple devices, each with its own memory & processing unit.
7) Host - Host is CPU, host initiates & manages execu^n of CUDA programs including data transfer.
8) Global memory - its the main memory space accessible by all threads in GPU, used for storing data that needs to be shared among threads
9) Shared memory -
10) CUDA Runtime API - Provides set of functions for managing device & memory & kernel execu^n, CUDA runtime API interact with CUDA enabled GPU's.

Teacher's Sign.: _____

CUDA Memory heirarchy



> Device pointers point to GPU memory, host pointer points to host memory

> Each thread has private local memory and shared memory of each thread is visible to all threads of same block.

> Global Memory - There's large amount of global memory. All running threads can read & write global memory and so can the CPU. The functions cudaMalloc, cudaFree, cudaMemcpy all deal with global memory.

> Constant Memory - It has its own cache. Not related to L1 & L2 cache of global memory. All threads have access to same constant memory but they can only read can't write. CPU sets values in constant memory before launching kernel

Teacher's Sign.:

> Shared Memory - Shared memory is very fast. Shared memory ~~uses~~ is used to enable fast communicaⁿ between threads in a block. Bank conflicts can slow down access down. it's fastest when all threads r read from different banks.

> Caches - There's also an L1 cache per multiprocessor and an L2 cache which ~~stores~~ shared between all multiprocessors.

> Local Memory - This is also part of main memory of the GPU so its ~~to~~ slow. Local memory is used automatically by threads when we run out of registers or when registers can't be used.

> Texture memory - It is specialized read-only memory optimized for texture fetching, suitable for texture mapping in graphics applicaⁿ.

CUDA-C kernel Function execuⁿ -

```
#include <stdio.h>
__global__ void multiply(int a, int b, int* res) {
    *res = a * b; }
int main() { int a=5, b=10;
  int *dev_res;
cudaMalloc((void **) &dev_res, sizeof(int));
multiply <<<1, 1>>> (a, b, dev_res);
int res;
cudaMemcpy(&res, dev_res, sizeof(int),
                cudaMemcpyDeviceToHost);
printf("%d * %d = %d\n", a, b, res);
cudaFree(dev_res);
return 0;
}
```

Teacher's Sign.: _____

1) Kernel declaration - The multiply function is declared as a CUDA kernel with the --global-- qualifier. This means it will run on the GPU and can be called from the CPU.
2) Kernel invocation - The kernel is launched from the CPU with one block and one thread (<<<1,1>>>) for only one thread is needed since we are multiplying two numbers.
3) Memory Access (*res = a * b) - The thread multiplies two input integers (a & b) and stores in the "res" in memory
4) Data Transfer - cudaMemcpy result copied from GPU to CPU after kernel execution
5) output - print output & free the memory cudaFree(dev_res).

Hello world.

```
#include <stdio.h>
--global-- void hellocuda(){
printf("hello world");}
int main(){
hellocuda <<<1,10>>>();
cudaDevicesynchronize()
return 0;
}
```

10 threads will print hello world.

```
nvcc hello-world.cu
./a.out
```

* Block Dimensions –
Block dimension refers to the number of threads per block. It defines size of a block which is a group of threads that execute the same kernel function. It is specified using a <numThreadperBlock> argument when launching.

* Grid Dimension – Refers to no. of blocks in grid. It defines the overall size of the grid, which consists of multiple blocks that execute same kernal function, specified using <numBlocks> argument when launching a kernel.

* CUDA-C++ for vector addition –

```
#include <cuda.h>
__global__ void Add_vec(float *a, float *b, float *C, int N)
{ int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i<N) {
        C[i] = a[i] + b[i]; } }
int main() { int N = 1024;
float* ha = (float*) malloc(N* sizeof(float));
float* hb = (float*) malloc(N* size of(float));
float* hc = (float*) malloc(N * sizeof(float));

float *da, *db, *dc;
cudamalloc(d.d, ha, N* sizeof(float)); cudaMem
cudamalloc(db, hb, N* size of(float));
cudamalloc(da, ha, N* sizeof(float), cudaMemcpyHostToDevice);
cudamalloc(db, hb, N*sizeof(float), cudaMemcpy HOSTTODevice);
  int ThreadsPerBlock = 256;
int Blocks = (N + ThreadsPerBlock - 1) / ThreadsPerBlock;
Vector Add_Vec <<< blocks, ThreadsPerBlock>>>
                (da, db, dc, N); Printf(h-c[i]);
cudaFree(da);
cudaFree(db);
cudaFree(dc); return 0; }.
```

Teacher's Sign: _____

```
#include <stdio.h>
#include <cuda.h>
__global__ void addInt(int *a, int *b, int *c) {
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i<1) { c[i] = a[i] + b[i]; }
}
int main() { int a=5, b=10, c;
int *d_a, *d_b, *d_c;
cudaMalloc(&d_a, sizeof(int));
cudaMalloc(&d_b, sizeof(int));
cudaMalloc(&d_c, sizeof(int));
cudaMemcpy(d_a, &a, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, sizeof(int), cudaMemcpyHostToDevice);

addInt<<<1,1>>>(d_a, d_b, d_c);
cudaMemcpy(&c, d_c, sizeof(int), cudaMemcpyDeviceToHost);
printf("Sum:", c);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0;
}
```

* Kernel launch - Launching kernel involves executing the kernel function on GPU with configuration. done by using <<< ... >>> syntax.
arguments -> 1) Grid size - Specifies no. of blocks in grid.
2) Block dimension - Specifies no. of threads per block
3) Dynamic shared memory - Allows allocation of shared memory per block dynamically.

Teacher's Sign.:

Thread Hierachy in Cuda memory

1) Threads - Individual unit of work executing the kernel function, Each thread has its own thread ID (threadIdx). can access, register, Shared memory & global memory.

2) Blocks - Groups of threads that cooperate & share data through shared memory. Each block has unique block ID (blockIdx)
- Threads within can synchronize their execution using --Syncthreads

3) Grid - collection of thread blocks launched on the GPU. Configured during kernel launch by specifying the no. of blocks in each dimension.

UNIT

HPC Applications

Parallel DFS -

1) DFS is a widely used algorithm for graph traversal essential in applications such as search & pathfinding. Parallel DFS involves exploring distinct branches of the search tree simultaneously, potentially yielding speedup.
2) Basic parallel DFS - Multiple threads or processes concurrently explore separate branches of the search tree, dividing the graph into subgraphs. This method is effective when branches are of similar size.
3) Workpile based parallel DFS - Using workpile data structure, multiple threads access nodes for explorat^n. Threads process nodes, expand neighbours & new nodes to the workpile. It avoids graph partitioning
4) Recursive parallel DFS - Threads are assigned subsets of the graph to explore, initiating DFS from specific nodes & recursively exploring adjacent nodes.
5) Task-based parallel DFS - Dynamically creating tasks or subproblems, threads independently explore assigned subgraphs. Threads generate tasks for unexplored nodes, dynamically scheduling them.

Parallel BFS -

1) BFS is a graph traversal algorithm that explores all vertices of a graph in breadth first order. Starts from a source node and visits neighbours before moving to their neighbors.
2) Level-Synchronous parallel BFS - The graph is divided into levels, with each level containing vertices at a specific distance from source. Multiple threads

Teacher's Sign.:

or processes work in parallel, each responsible for visiting vertices at a particular level. Initially source vertex is assigned to 1st level & subsequent levels are processed in parallel.

3) Workpile -based parallel BFS - Similar to parallel DFS, a workpile data structure manages nodes to be explored. Multiple threads access workpile concurrently, processing nodes until its emptly.

4) Recursive parallel BFS - Threads are assigned subsets of graphs, initiating BFS from specific nodes & exploring adjacent nodes recursively

5) Hybrid approach -

Parallel Bubble Sort -

1) Parallalized version of classic bubble sort algorithm It involves dividing the sorting process among multiple threads or processes to speed up operation.

2) Divide & Conquer - In parallel bable Sort, the list to be sorted is divided into smaller sublists, and each sublist is sorted independently by threads

3) Parallel Comparison & Swapping - Each thread or process compares adjacent elements within its sublist and swaps them if necessary. Similar to sequential bubble sort.

4) Synchronizaⁿ - Synchronizaⁿ mechanisms are used to ensure that threads or processes coordinate their sorting operaⁿ correctly.

5) Combining Results - once all sublists are sorted independently, the sorted sublists are combined or merged to produce final list (sorted).

Teacher's Sign.: _____

Odd-even Transportation in Seq. bubble sort.

	1	2	3	4	5	6	7	8
odd phase.	3	2	3	8	5	6	4	1

Start from odd index and pair with immediate next element in sequence. and then swap according to condition.

	1	2	3	4	5	6	7	8
even phase	2	3	3	8	5	6	1	4

start from even index and pair with immediate next. and swap if necessary.

odd phase.	2	3	3	5	8	1	6	4
even phase.	2	3	3	5	1	8	4	6
odd phase.	2	3	3	1	5	4	8	6
even phase.	2	3	1	3	4	5	6	8
odd phase	2	1	3	3	4	5	6	8

swap. swap.

So 1 2 3 3 4 5 6 8. → Sorted array

In sequential bubble sort it is working 1 element per process.

Parallel Formula of odd-even Transportation

Assume there are Two threads Thread 1 & Thread 2 and an array

arr = [3, 1, 4, 2, 5]
(1, 2, 3, 4, 5)

> In odd phase Thread 1 is assigned to arr[1] and Thread 2 is assigned to next odd index ie arr[3] In odd phase Thread assigned to odd index.

Thread 1 has arr[1] = 3 it compares with immediate next element in array ie arr[2] Compares their values & swaps them if necessary

So Thread 1 compares 3 with 1

(arr[1] > arr[2], swap) we get [1,3,4,2,5]

Same for Thread 2

Thread 2 compares 4 with 2

(arr[3] > arr[4], swap) we get.

[1, 3, 2, 4, 5]
(1, 2, 3, 4, 5)

> In Even phase Thread is assigned to even index.

So Thread 1 is assigned to arr[2] compares with arr[3] & Thread 2 is assigned to arr[4] compares with arr[5]

Thread 1 compares 3 with 2

(arr[2] > arr[3], swap] we get.

[1, 2, 3, 4, 5].

Thread 2 compares 4 with 5.

(arr[4] < arr[5], no swap] we get.

[1, 2, 3, 4, 5] → Sorted array.

Time cost = $\Theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \Theta(n) + \Theta(n)$.

Teacher's Sign.: _____

4) Parallel merge sort -

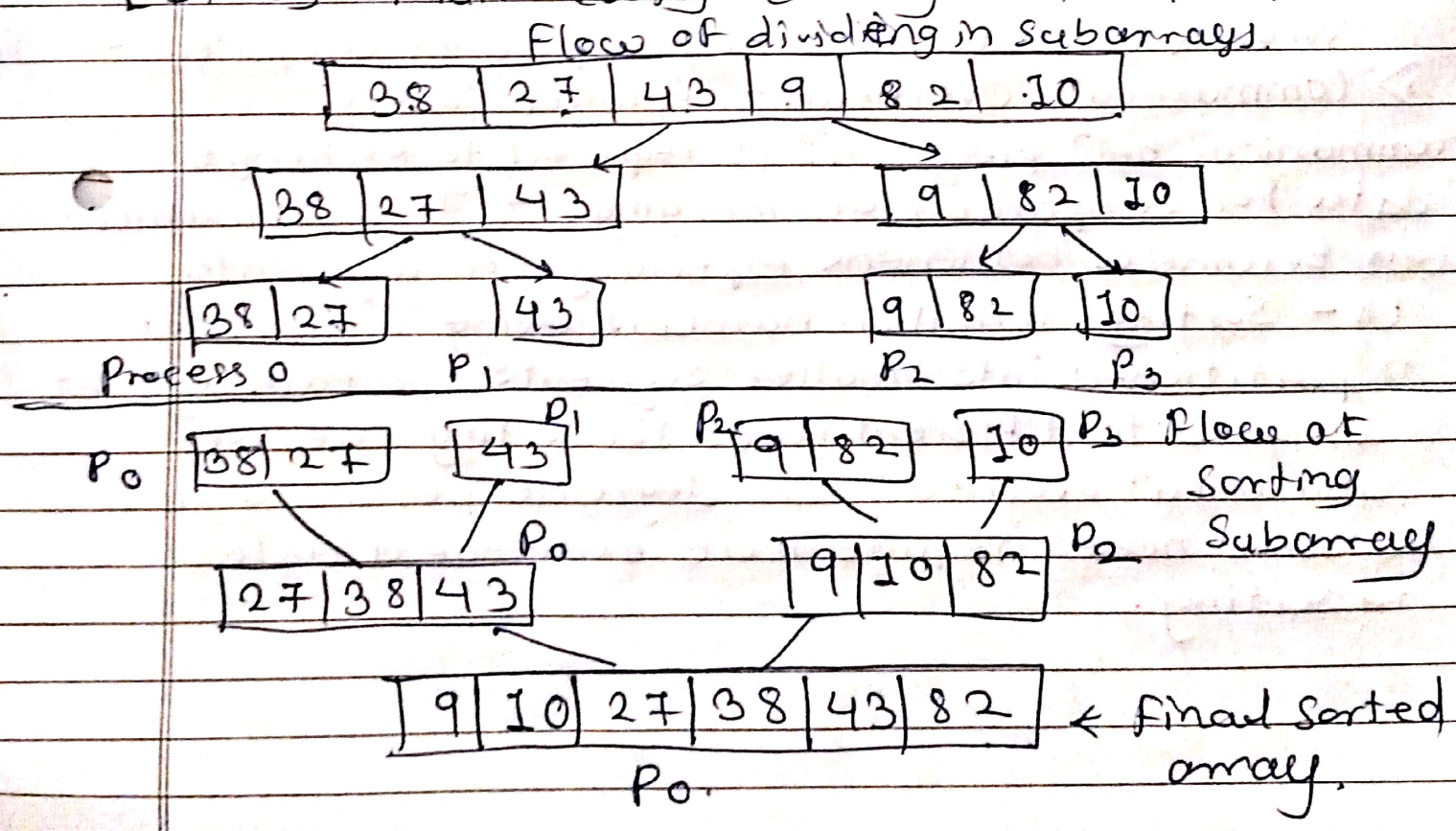
1) Parallalized version of sequential merge sort to improve performance on multicore processors Sorting process is divided among multiple threads

2) Consider unsorted array [38, 27, 43, 9, 82, 10]

2) Initial Splitting (Divide phase) -
Divide array into smaller subarrays
It can be performed in parallel by dividing input array among multiple threads.

3) Sorting - Sort each subarray idependently using parallel ~~sorting~~ programming model like openMP or MPI, to sort each half recursively.

4) Merge - Once both halves are sorted they need to be merged into single sorted list.
This can be done sequentially by any processor.
For example. two processors can merge [27, 38] & [3, 43] simultaneously resulting [3, 27, 38, 43].



Teacher's Sign.: _____

C++

```
void parallel_merge(int* data, int low, int high){
  if (low < high){
    int mid = (low + high)/2;
  #pragma omp parallel sections num_threads(2)
  { #pragma omp section
    { parallel_merge(data, low, mid); }
  #pragma omp section
    { parallel_merge(data, mid+1, high); }
  }
  merge(data, low, mid, high);
  }
}
void merge(int* data, int* left, int* right)
{ //function of merging
}
```

What are issues in Sorting on parallel computers -

1) Communica^n overhead - Parallel Sorting, communica^n bet^n processors is required to exchenge data for comparison or merging. - This communica^n can become a bottleneck, especially for large data.

Ex - Sorting a million numbers, ~~string~~ The dataset is partitioned into smaller segments and each segment is assigned to different nodes for sorting. Excessive communica^n overhead can degrade performance as some nodes require more exchange of data for sorting.

2) Load balancing - The size of subproblems to assigned to processors affects performance if subproblem are too small, the overhead of managing threads can outweigh the benefits of parallalizaⁿ
Ex - while performing sorting on large dataset dataset partitioned into smaller segments due to variation is the sizes of segments or different processing speeds between nodes. Some nodes may finish earlier and may remain idle.

3) Synchronizaⁿ challenges - Parallel sorting algo often require Synchronizaⁿ points where processors need to wait for each other before processing
ex - Odd-even Transportaⁿ sort, thread need to compare elements with neighbors, if not properly Sync threads might access the same data.

4) Limited parallelism in sorting Algo -
Some sorting algo have inherent limitations when parallalized. ex - insertion sort.

Kubernets - It is an open source container orchestration platform, It automates deployment, scaling & management of containerized applicaⁿ
Features.
1) Container - Kubernets automates the deployment Scaling & management of containerized apps.
2) Scalability - Easily scale applicaⁿ up or down based on demand Kubernets can automatically allocate resources & manage containers

Teacher's Sign.

3) Self healing - If container crashes, kubernets automatically restarts it.
4) Load balancing - Kubernets can distribute incoming traffic across multiple instances of applicon.
5) Storage - Integrates with vantus storage solutions ~~states~~ allocating persistent storage for containarized apps.

Application.
1) Modern web Apps
2) Cloud Native Dev
3) Machine learning
4) HPC

Application of GPU
1) Scientific computing
2) Machine learning
3) Data Science Analytics
4) Media & Entertainment.
5) AI
6) Forecasting