

LP 5 Viva

DL

A1:

- The purpose of using Artificial Neural Networks for Regression over Linear Regression is that the linear regression can only learn the linear relationship between the features and target and therefore cannot learn the complex non-linear relationship. In order to learn the complex non-linear relationship between the features and target, we are in need of other techniques. One of those techniques is to use Artificial Neural Networks. Artificial Neural Networks have the ability to learn the complex relationship between the features and target due to the presence of activation function in each layer. Artificial Neural Networks are one of the deep learning algorithms that simulate the workings of neurons in the human brain.

A2:

Binary classification using Deep Neural Networks Example:

Classify movie reviews into "positive" reviews and "negative" reviews, just based on the text content of the reviews. Use IMDB dataset.

- The output layer uses `sigmoid` (for binary) or `softmax` (for multi-class).
- **Optimizer:** Algorithm that adjusts weights to reduce loss(**Adam = Adaptive Moment Estimation**)

- Each row is a review, and columns represent whether a word (index) is present (1) or not (0).
- Example: `X_train.shape = (25000, 10000)` (25k reviews × 10k words).

A3: Convolutional neural network (CNN)

Use MNIST Fashion Dataset and create a classifier to classify fashion clothing into categories

- To build a neural network that classifies **Fashion MNIST** grayscale images (28×28 pixels) into **10 categories** like *T-shirt, Trouser, Dress, etc.*
- Pixel values range from 0–255. Normalize to 0–1 for better training performance.
- `plot_image` : Displays image and predicted label vs actual label (blue if correct, red if wrong).
- `plot_value_array` : Shows bar chart of prediction probabilities.
- **CNN (Convolutional Neural Network)** is a type of **deep learning model** that is especially powerful for **image recognition and classification tasks**. It mimics how the human brain sees and processes images.

Key Layers of a CNN:

Layer Type	Purpose
Convolution Layer	Detects features like edges and patterns by sliding filters over the image.
ReLU Layer	Applies non-linearity to make the network learn complex patterns.
Pooling Layer	Reduces size and complexity while keeping important info (e.g., max pooling).
Flatten Layer	Converts the 2D matrix into a 1D vector for the dense layer.
Dense (Fully Connected)	Makes final classification decisions based on learned features.

A4: RNN

To study and understand Recurrent Neural Network by doing an analysis and designing prediction systems.

Use the Google stock prices dataset and design a time series analysis and prediction system using RNN.

- A **Time Series** is a sequence of **data points collected or recorded at specific time intervals** — like one after the other.
- So, Time Series Analysis is all about understanding patterns in that data over time
- **Recurrent Neural Network** is a type of neural network that is **designed to work with sequences of data**. (Eg: Weather data per day)
- RNNs see one input at a time but also remember what came before it!
- A Recurrent Neural Network(RNN) is a type of Neural Network where the output from the the previous steps are fed as input to the current step.
- The goal is to **predict future stock prices (GOOG)** using **past price trends** with an **LSTM neural network**, which is a type of **Recurrent Neural Network (RNN)** perfect for sequence data like stock prices over time.
- **How many dimensions must the inputs of an RNN layer have?**
 - 🖱️ **3 dimensions: (batch_size, time_steps, features_per_step)**

Alright buddy! Here's a **clear and direct** answer key for your **viva** — everything simple, neat, and to the point 🔥:

Assignment 1: Linear Regression using Deep Neural Networks

32. Architecture of DNN model for regression:

- Input Layer → Hidden Layers (with ReLU activation) → Output Layer (single neuron, linear activation)

33. Why use DNN over simple linear regression?

- DNN can capture **complex non-linear relationships** which simple linear regression cannot.

34. Advantages of ReLU over Sigmoid in hidden layers:

- Avoids vanishing gradient problem
- Faster computation
- Better performance during training

35. Initialization technique used for weights:

- **He Initialization** for ReLU activations (or Xavier for general)

36. If learning rate is too high or too low:

- **Too high** → Model won't converge, jumps around
- **Too low** → Model trains very slowly

37. Role of batch size during training:

- Controls how many samples are processed before updating the model weights.
- Smaller batches → Noisier updates (can generalize better)
- Larger batches → Smoother updates, faster computation

38. Regularization in linear regression models:

- Reduces overfitting by adding penalty terms (L1 or L2) to the loss function.

39. Effect of too many hidden layers on simple regression problem:

- Overfitting
- Increased training time without much gain

40. How to validate model's performance:

- Use a **test set** or **cross-validation**

- Evaluate using metrics like **Mean Squared Error (MSE)**

41. Risks of underfitting in linear regression:

- Model cannot capture underlying patterns → Poor accuracy
-

Assignment 2: Classification using Deep Neural Networks

1. Architecture of classification DNN:

- Input Layer → Hidden Layers (ReLU) → Output Layer (Softmax for multi-class)

2. How to encode class labels:

- Use **One-hot encoding** for multi-class classification

3. What is softmax activation:

- Converts output scores into **probabilities** that sum to 1.

4. What is categorical cross-entropy loss:

- Measures the difference between true and predicted probability distributions for classification.

5. How dropout helps:

- Randomly drops neurons during training → Reduces overfitting

6. Handling overfitting in classification models:

- Dropout, Early stopping, Data augmentation, L2 Regularization

7. What is confusion matrix:

- A table that shows True Positives, True Negatives, False Positives, False Negatives
- Helps visualize performance

8. Dealing with unbalanced datasets:

- Oversampling, Undersampling, Using weighted loss functions

9. What is data augmentation:

- Artificially increasing dataset by flipping, rotating, zooming images

10. What are learning rate schedules:

- Methods to change the learning rate during training → Helps faster and stable convergence
-

Assignment 3: CNN for Fashion MNIST Classification

1. Why CNNs are better for images:

- CNNs **capture spatial hierarchies** using filters → More efficient than fully connected layers

2. Size of filter used and why:

- 3×3 or 5×5 → Small enough to detect fine features but big enough to capture patterns

3. What is feature map in CNN:

- The output of a convolutional layer showing detected features.

4. Concept of receptive field in CNN:

- The region of input image that affects a neuron's activation

5. How CNN handles translation invariance:

- **Shared weights** and **pooling layers** help recognize objects despite slight movements.

6. Why max pooling over average pooling:

- **Max pooling** captures the **strongest features** (edges, textures)

7. Optimizer chosen and why:

- **Adam optimizer** → Fast convergence, adaptive learning rates

8. Vanishing gradients effect on CNNs:

- Makes deep networks hard to train because gradients shrink too small for weight updates.

9. What are Batch Normalization layers:

- They normalize activations → Faster training and better stability

10. Tuning hyperparameters:

- Using validation set, try different **learning rates**, **epochs**, **batch sizes**, etc.
-

Assignment 4: Mini Project - Human Face Recognition

1. Preprocessing steps:

- Face detection, resizing, normalization

2. CNN architecture used:

- Pre-trained models like **FaceNet**, **VGG-Face**, or **custom CNNs**

3. What is face embedding:

- A numerical representation (vector) of a face image used for comparison

4. How Triplet Loss works:

- Ensures that **anchor** and **positive** (same person) are closer than **anchor** and **negative** (different person)

5. Classification vs Verification in face recognition:

- **Classification**: Assign to a known identity
- **Verification**: Compare if two faces are the same

6. Handling pose and lighting variations:

- Data augmentation, use robust pre-trained models

7. What is one-shot learning:

- Model learns to recognize a class from **one example** → Useful in face recognition

8. Data augmentation techniques:

- Flip, crop, rotate, brightness variation

9. Deploying face recognition into mobile app:

- Convert model to **TensorFlow Lite** or **ONNX**, optimize for mobile hardware

10. Improvements using transfer learning:

- Use a **pre-trained network** and fine-tune on your specific face dataset
-

Extra Questions

11. What is Batch Size?

- Number of samples processed before model updates weights once.

12. What is Dropout?

- Randomly turning off neurons during training to prevent overfitting.

13. What is RMSprop?

- Optimizer that adjusts the learning rate for each parameter individually.

14. What is Softmax Function?

- Turns outputs into a probability distribution (sums to 1).

15. What is ReLU Function?

- Activation function: $f(x) = \max(0, x)$ → Makes model non-linear, fast computation.
-

16. What is Binary Classification?

- Predicting two classes (0 or 1) like spam/not spam.

17. What is Binary Cross Entropy?

- Loss function for binary classification measuring difference between predicted and actual labels.

18. What is Validation Split?

- Percentage of training data reserved for validation during training.

19. What is Epoch Cycle?

- One complete pass through the entire training dataset.

20. What is Adam Optimizer?

- Optimizer combining advantages of **RMSprop** and **Momentum** → Fast and efficient
-

21. What is Linear Regression?

- Predicting a continuous value by fitting a straight line to data.

22. What is a Deep Neural Network?

- A neural network with multiple hidden layers between input and output.

23. Concept of Standardization:

- Scaling data to have **zero mean and unit variance**.

24. Why split data into train and test?

- To evaluate model performance on **unseen data**.

25. Application of Deep Neural Networks:

- Image recognition, Speech processing, Healthcare diagnosis, Self-driving cars, Chatbots
-

26. What is MNIST dataset for classification?

- Dataset of handwritten digits (0-9) images (28×28 pixels).

27. How many classes in MNIST?

- 10 classes (digits 0 to 9)

28. What is 784 in MNIST?

- Each image is 28×28 pixels → 784 total input features (28×28=784)

29. How many epochs in MNIST training?

- Typically **10 to 50 epochs** depending on model

30. What are hardest digits in MNIST?

- 4 vs 9, 3 vs 5, and 7 vs 9 are hardest to classify.
-

31. What is Exploratory Analysis?

- Analyzing datasets to summarize their main characteristics with visual methods.

32. What is Correlation Matrix?

- A table showing correlation coefficients between variables.

33. What is Conv2D used for?

- It applies 2D convolution over input images in CNNs → extracts features like edges, patterns.

✅ You're all set buddy for your viva!

Would you also like a **small PDF notes version** for easy revision? 📄✨ (I can quickly generate it if you want!)

HPC

HPC 1

Design and implement Parallel Breadth First Search and Depth First Search based on existing

algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS .

- BFS (Breadth-First Search) and DFS (Depth-First Search) are **two graph traversal algorithms**. BFS explores a graph by visiting all nodes at a given level before moving to the next, while DFS explores the graph by going as deep as possible along a branch before backtracking.

Key Differences:

Feature	BFS	DFS
Search Strategy	Explores nodes level by level	Explores one path deeply before backtracking

Shortest Path	Guarantees shortest path in unweighted graphs	Does not guarantee shortest path
Memory Usage	More memory-intensive (uses a queue)	More memory-efficient (uses a stack)
Applications	Shortest path, connectivity, searching for nearby nodes	Topological sorting, cycle detection, memory-constrained scenarios

- Both BFS and DFS have time complexity $O(V + E)$, where V is vertices and E is edges.
- Parallelism improves performance by allowing multiple nodes to be explored simultaneously, reducing overall search time.
- How is a visited array managed safely in parallel BFS?
 - **Visited array** keeps track of which nodes are already visited.
 - In parallel BFS, **each thread must lock or atomically update** the visited array safely to **avoid two threads visiting the same node**.
 - A lock ensures only one thread can access or update a shared data at a time. In BFS, locking helps safely update the visited array to avoid multiple threads visiting the same node.
- **OpenMP(Open Multi-Processing)** is a **library** that helps **add parallelism** in **C, C++, or Fortran** programs easily. It uses simple directives to split work among threads and make programs faster
- **Critical sections** are blocks of code that **only one thread** can execute at a time.
Prevents data corruption when multiple threads access **shared data**.
- Parallelizing DFS causes issues like **race conditions**(many threads might try to visit the same node at the same time), **duplicate work**(If multiple threads start their own deep searches, they might visit the same parts again and again unnecessarily), and **wrong traversal order**(DFS is supposed to go deep along one branch first before trying another.In parallel, multiple branches may get explored at the same time.So, you lose the correct depth-first order) due to deep dependency between nodes.

- **Breadth-wise parallelism** in BFS means exploring all nodes at the same level simultaneously using multiple threads.
- **Without synchronization**, multiple threads may visit the same node and corrupt the data, leading to wrong or inconsistent traversal results.
- We use `#pragma omp parallel for` to **parallelize a for loop** in OpenMP.

What are advantages and disadvantages of parallel BFS?

Advantages:

- Faster traversal on big graphs.
- Good for level-wise work.

Disadvantages:

- Synchronization overhead.
- Load imbalance if some nodes have more neighbors.

How Many Threads Are Being Used?

- **By default, OpenMP** will use **as many threads as there are CPU cores** available.
- Unless you explicitly set the number of threads using:

```
omp_set_num_threads(number);
```

it uses the system's default maximum.

Example: If your system has 8 cores → it will use **8 threads**.

Types of Scheduling Policies:

1. Static Scheduling

- **Work is divided equally** among threads **before** execution starts.
- Each thread **already knows** what part it will work on.
- Best when each part of the work **takes the same time** (uniform work).

✓ Advantages:

- Very **low overhead**.
- Fast and simple.

✗ Disadvantages:

- **Bad** if tasks take **different times** → Some threads may finish early and sit idle (load imbalance).
-

2. Dynamic Scheduling

- **Work is divided during execution.**
- A thread takes a chunk of work **when it finishes** the previous chunk.
- Good when tasks are **unpredictable** (different work times).

✓ Advantages:

- **Better load balancing.**
- Threads stay busy.

✗ Disadvantages:

- Slightly **more overhead** (needs management during runtime).
-

3. Guided Scheduling

- Similar to dynamic, but...
- **Big chunks** of work are given **first**, and **smaller chunks** later.
- Initially gives large pieces, then reduces to small ones to finish smoothly.

HPC 2

Parallel Bubble Sort and Merge Sort using OpenMP.

- **Grain size** = how much work a thread is given at once.
- If grain size is too small, there will be too much overhead in managing work between threads. If it is too large, load imbalance happens and some threads become idle.

So, an optimal grain size is important for best performance.

Grain Size	Problem
Too Small	High overhead (too much managing)
Too Large	Load imbalance (some threads idle)

How does thread scheduling affect performance in sorting?

- Good scheduling = **Balanced load** among threads = Faster sort.
- Bad scheduling = Some threads may finish early and sit idle while others are still working.
- **Dynamic scheduling** is often better for unpredictable workloads (like varying sizes of partitions).

What is False Sharing in parallel Sorting?

When multiple threads **access different variables** that **happen to be stored close together** in memory (like in the same *cache line*),

even if they don't actually share data,

➡ **it still causes performance problems!**

Because in CPUs, memory is fetched in **blocks** called **cache lines** (typically 64 bytes).

If one thread **writes** to a variable in a cache line, the **whole cache line** gets **invalidated** for other threads, even if they are working on totally different variables inside it!

This forces **unnecessary communication** between threads → called **False Sharing**.

- **Merge Sort:**
 - **Divide:** It divides the array into two halves recursively until each part has a single element.
 - **Conquer:** It then **merges** those smaller sorted parts back together to form the final sorted array.

How do you divide the array for parallel Merge Sort?

- You **split** the array into **two halves** (or more depending on the number of threads).
- Each half is **sorted independently** in parallel.
- After sorting, **merge** the halves carefully.

What is memory overhead in Merge Sort and how do you manage it?

- **Merge Sort needs extra memory** to store temporary arrays during merging.
- Overhead $\approx O(n)$ additional space.
- Solution: Instead of creating a new temporary array again and again at every merge step, → create one big temp array at the beginning and pass it everywhere

- **Bubble Sort:**

It is the kind of opposite to selection sort here we compare two elements [j, j+1] and swap if

$arr[j] > arr[j+1]$. By doing this last element always stay sorted and the max automatically gets placed at last.

```
#include <bits/stdc++.h>
void bubbleSort(vector<int>& arr, int n)
{
    int didSwap = 0;
    for(int i = n-1; i >= 0; i--)
    {
        for(int j = 0; j <= i-1; j++)
        {
            if(arr[j] > arr[j + 1])
            {
                swap(arr[j], arr[j + 1]);
                didSwap = 1;
            }
        }
    }
}
```

```

    }
    if(didSwap == 0)
    {
        break;
    }
}
}

```

TC: Average and Worst $\rightarrow O(n^2)$

Best $\rightarrow O(n)$

The **worst-case** time complexity of **parallel Bubble Sort is $O(n^2)$** . This is because even with parallel processing, Bubble Sort still requires nested loops for comparison and swapping, leading to the same quadratic time complexity as the sequential version

HPC 3

Implement Min, Max, Sum and Average operations using Parallel Reduction.

- **Parallel Reduction** means breaking a large operation (like sum, max, min) into smaller parts, processing them simultaneously on multiple threads, and then combining the results.
- Associative operation ensures combining partial results in any order still gives the correct final answer.
- **What does the reduction clause in OpenMP do internally?**
 - In OpenMP,
Creates
private copies of the reduction variable for each thread, each thread computes its part, and then OpenMP combines all the results safely at the end.
- **Why might a parallel reduction be slower than serial for small datasets?**
 - **Overhead of creating threads** and **synchronization costs** are high.

- For small data, the **time spent managing threads** is **more** than the actual computation time.
 - So serial execution can actually be faster for small datasets.
 - **Cache coherence** means keeping copies of shared data consistent across CPU caches.
 - **Atomic operation** is a *single, uninterruptible* memory operation.
 - Example: incrementing a shared counter safely.
 - **Reduction** is a *collective operation* where many partial results are combined.
 - To reduce communication overhead, we use thread-local variables, minimize synchronization, and combine results in **hierarchical fashion** (tree-based reduction).
- **Parallel version** → **OpenMP** used with **16 threads** for faster computation.

HPC 4

Write a CUDA Program for :

1. Addition of two large vectors
2. Matrix Multiplication using CUDA C

👉 **CUDA** stands for **Compute Unified Device Architecture**.

It is a

technology developed by NVIDIA that allows you to **use the GPU (Graphics Processing Unit) for general-purpose computing** — **NOT** just for gaming or rendering graphics, but also for calculations like matrix multiplication, simulations, machine learning


- Normally your **CPU** does all the heavy math and logic in your computer.

- But **CPU has fewer cores** (like 4, 8, or 16).
- **GPU has thousands of tiny cores** that can work **in parallel** — meaning **super fast** for big tasks!
- **CUDA** lets you **write C/C++ code** that **runs on your GPU** directly.

Structure of a CUDA Program

A typical CUDA program has these steps:

- **Allocate memory** on the **host (CPU)** and **device (GPU)**.
- **Copy data** from **host → device**.
- **Launch kernel** (GPU function) to do computations.
- **Copy results** back from **device → host**.
- **Free memory**.

 **Important:** A **kernel** is a special function that runs on the GPU. to do computations.

The syntax to launch a kernel is:

```
kernel_name<<<num_blocks, threads_per_block>>>(kernel_arguments);
```

- `num_blocks` : How many blocks you want.
- `threads_per_block` : How many threads inside each block.

GPU memory is allocated using: `cudaMalloc(void **ptr, size_t size)` function

```
int *d_arr;
cudaMalloc(&d_arr, n * sizeof(int));
```

Global, Shared, and Local Memories in CUDA

Type	Scope	Speed	Lifetime
Global	Accessible by all threads	Slow (compared to shared)	Whole program

Shared	Shared by all threads in a block	Fast	Lifetime of block
Local	Private to each thread	Slow (actually stored in global memory if needed)	Lifetime of thread

How is Thread Indexing Done in a CUDA Kernel?

`threadIdx.x`, `threadIdx.y`, and `threadIdx.z` represent the index of a thread within a block along the x, y, and z dimensions.

Calculating a Unique Thread Index (Global Index):

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

- `threadIdx.x` → thread's index inside block.
- `blockIdx.x` → which block.
- `blockDim.x` → number of threads in a block.

What Happens If Too Many Threads are Launched in CUDA?

- GPU may **fail to launch** the kernel.
- **Performance will degrade** badly.
- **Kernel launch error** can occur (`cudaErrorLaunchFailure`).
- Threads beyond hardware limits simply **won't run**.

 Each GPU has limits on:

- Threads per block
- Threads per multiprocessor

What is Warp Size in CUDA?

- a warp is **a group of 32 threads that execute the same instruction concurrently**.

How Does Coalesced Memory Access Improve Performance?

If **consecutive threads** access **consecutive memory addresses**, it's called **coalesced memory access**.

✓ Benefits:

- Fewer memory transactions.
- Increased Bandwidth Utilization
- Faster Execution

What are CUDA Streams and Why Are They Useful?

In CUDA, streams are **sequences of commands executed on the GPU, allowing for asynchronous and concurrent execution of operations.**

They enable efficient resource utilization by overlapping tasks like kernel execution and memory transfers. Streams are crucial for optimizing performance and leveraging the parallelism of GPUs.

Explain How Grid-Stride Loops Help in CUDA Programming

In case the **total number of elements > number of threads**, **grid-stride loops** allow a **single thread to process multiple data elements**, enabling scalability for large datasets even when the total number of threads in a grid is limited.

- Same code works for small or huge arrays.

```
cudaMalloc(&dev_A, size);                // Allocates space on GPU
cudaMemcpy(dev_A, A, size, cudaMemcpyHostToDevice); // Copies CPU data to GPU
```

HPC Mini

I used

task parallelism through Python's `concurrent.futures.ThreadPoolExecutor` to run recursive quicksort calls in parallel.

1. Applications of Parallel Computing

- Weather forecasting
 - Image processing (e.g., MRI scans)
 - Scientific simulations (e.g., nuclear research)
 - Big Data analysis (e.g., Google Search)
 - Video rendering and gaming
-

2. Basic Working Principle of VLIW Processor

- **VLIW = Very Long Instruction Word**
 - Packs multiple operations into one long instruction.
 - Executes them **simultaneously** without complex hardware for instruction scheduling.
 - Compiler handles instruction parallelism, not the processor.
-

3. Control Structure of Parallel Platform

- **Control structure** decides **how tasks are assigned and executed**.
 - Two types:
 - **Centralized control** (one master controls everything)
 - **Distributed control** (each processor makes local decisions)
-

4. Basic Working Principle of Superscalar Processor

- Fetches **multiple instructions per clock cycle**.
 - Tries to execute **independent instructions** at the same time.
 - Needs **dynamic instruction scheduling**.
-

5. Limitations of Memory System Performance

- Memory latency (slow speed compared to CPU)
- Bandwidth limitations

- Cache misses
- Contention (many cores accessing same memory)

6. SIMD, MIMD & SIMT Architecture

Type	Full Form	Example
SIMD	Single Instruction Multiple Data	GPUs, vector processors
MIMD	Multiple Instruction Multiple Data	Multi-core CPUs
SIMT	Single Instruction Multiple Threads	Modern GPUs (NVIDIA CUDA)

7. Types of Dataflow Execution Model

- **Static dataflow** (fixed paths)
- **Dynamic dataflow** (paths decided at runtime)
- **Token dataflow** (tokens carry data and control)

8. Short Notes on UMA, NUMA & Level of Parallelism

- **UMA**: Uniform Memory Access, all processors access memory at same speed.
- **NUMA**: Non-Uniform Memory Access, access time depends on memory location.
- **Levels of Parallelism**: Bit-level, instruction-level, task-level, and process-level.

9. Cache Coherence in Multiprocessor System

- Ensures all caches have **consistent view** of memory.
- Example: **MESI protocol** (Modified, Exclusive, Shared, Invalid)

10. N-wide Superscalar Architecture

- Can fetch and execute **N instructions per cycle**.
- For example, a 4-wide superscalar can execute 4 instructions at once.

11. Interconnection Network and Types

- Connects processors and memories.
 - Types:
 - Bus
 - Ring
 - Mesh
 - Hypercube
 - Crossbar
-

12. Short Note on Communication Cost

- Time taken to **send/receive data** between processes.
 - Includes **latency + transfer time**.
-

13. Compare Write Invalidate vs Write Update Protocol

	Write Invalidate	Write Update
Working	Invalidate other copies first	Update all copies immediately
Bandwidth	Low	High
Example	MESI protocol uses invalidate	

14. Decomposition, Task & Dependency Graph

- **Decomposition:** Break a big problem into smaller parts.
 - **Task:** A piece of work.
 - **Dependency Graph:** Shows how tasks depend on each other.
-

15. Granularity, Concurrency & Task Interaction

- **Granularity:** Size of tasks.
- **Concurrency:** Tasks running at the same time.

- **Task Interaction:** Communication between tasks.
-

16. Decomposition Techniques

- **Domain decomposition** (divide data)
 - **Functional decomposition** (divide functions)
-

17. Characteristics of Task and Interactions

- Task size
 - Communication needs
 - Dependency between tasks
-

18. Mapping Techniques

- Assigning tasks to processors.
 - Types:
 - Static Mapping
 - Dynamic Mapping
 - Load Balancing Mapping
-

19. Parallel Algorithm Model

- Models like **PRAM** (Parallel Random Access Machine), **BSP** (Bulk Synchronous Parallel), **Work-Depth Model**.
-

20. Thread Organization

- Group of threads executing in parallel.
 - **Thread blocks** (CUDA), **pthread** (POSIX).
-

21. Short Note on IBM CBE

- **Cell Broadband Engine.**

- Used in **PlayStation 3**.
 - Powerful for scientific applications.
-

22. History of GPUs and NVIDIA Tesla GPU

- GPUs started for graphics.
 - Tesla GPUs introduced **GPGPU** (General Purpose GPU computing).
-

23. Broadcast & Reduce Operation with Diagram

- **Broadcast**: Send data from one to many.
 - **Reduce**: Collect data from many to one.
 - (I can draw a quick diagram if needed!)
-

24-25. Broadcast and Reduction on Ring

- **One-to-all broadcast**: Send data around a ring.
 - **All-to-one reduction**: Collect data around a ring.
-

26. Pseudo Code for One-to-all Broadcast on Hypercube

```
for each dimension d:  
  if my_id has 0 at dth bit:  
    send data to node with 1 at dth bit  
  else:  
    receive data
```

27. All-to-all Broadcast & Reduction

- **Linear array**: Neighbors share data.
 - **Mesh**: Rows then columns broadcast.
 - **Hypercube**: $\log(n)$ steps broadcast.
-

28. Scatter and Gather Operation

- **Scatter:** Break data and send to different processors.
 - **Gather:** Collect parts of data from processors.
-

29. Circular Shaft on Mesh and Hypercube

- **Circular Shaft:** Rotating communication paths across Mesh or Hypercube.
-

30. Different Approaches of Communication

- Direct messaging
 - Shared memory
 - Collective communication (MPI)
-

31. All-to-all Personalized Communication

- Each processor sends **unique messages** to all others.
-

Phew! 🤔

That's all 31 answered shortly for your viva — and you are totally ready now, buddy! 🚀

If you want, I can also give you a **quick 10-minute revision sheet** 📄 for this entire syllabus!

Would you like that? 🎯 (say yes!)