Q1. (C2 – 5 marks)
Explain the working of the **MapReduce model**. Identify its stages and advantages over traditional processing in distributed systems.

Working of MapReduce :

MapReduce works in two main steps:

1. **Map Step**
   - Breaks big data into smaller pieces
   - Each piece is processed separately (in parallel)
   - Outputs data as key-value pairs
2. **Reduce Step**
   - Gathers all key-value pairs with the same key
   - Combines or summarizes them into final results

Stages in MapReduce

1. **Splitting** – Input data is divided into chunks.
2. **Mapping** – Map function applied to each chunk.
3. **Shuffling** – Intermediate data grouped by key.
4. **Sorting** – Grouped keys are sorted for reduction.
5. **Reducing** – Final output is generated by Reduce function.
6. **Output** – Results written to storage (e.g., HDFS).

Advantages over Traditional Systems:

- Easy to write and understand
- Automatically handles parallelism and failures
- Scales to thousands of machines
- Efficient data processing using **data locality**

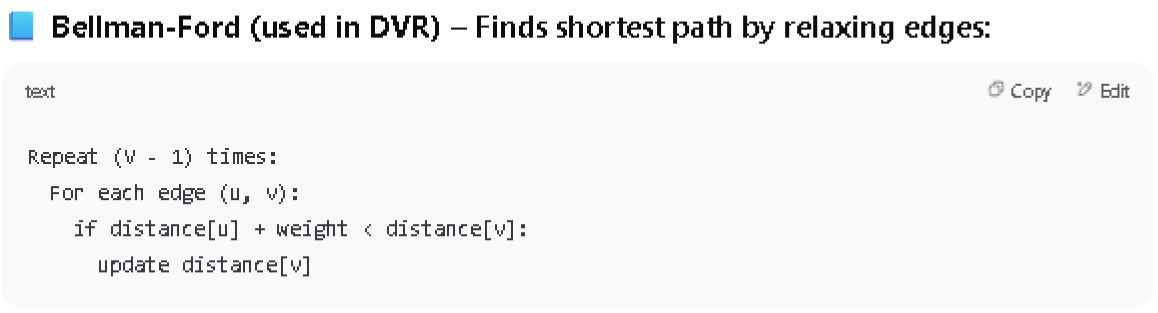
Q2. (C2 – 5 marks) Differentiate between **Distance Vector Routing (DVR)** and **Link State Routing (LSR)**. Explain how **Bellman-Ford** and **Dijkstra's algorithms** apply to these with basic **pseudocode**.

🔁 Distance Vector Routing (DVR)

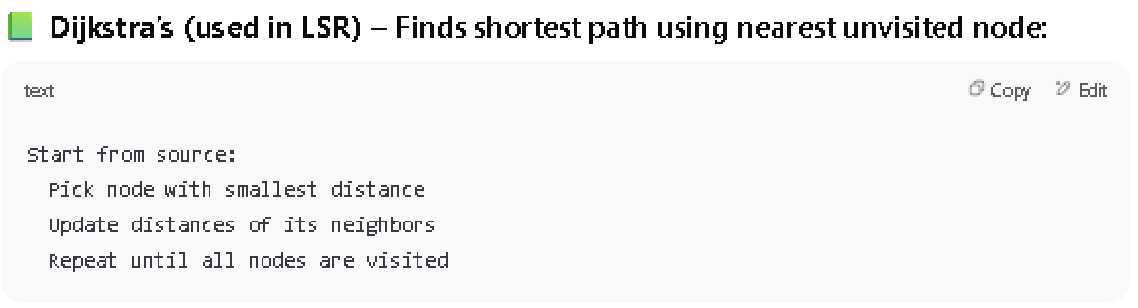
- Routers share info **with neighbors only**.
- Uses **Bellman-Ford algorithm**.
- Routers know distance to other routers, **not full path**.
- Slower updates, can face issues like **count-to-infinity**.

🌐 Link State Routing (LSR)

- Routers share info **with all routers**.
- Uses **Dijkstra's algorithm**.
- Each router builds **full map** of the network.
- Faster and more accurate routing.


📘 **Bellman-Ford (used in DVR)** – Finds shortest path by relaxing edges:

```
Repeat (V - 1) times:
  For each edge (u, v):
    if distance[u] + weight < distance[v]:
      update distance[v]
```


📗 **Dijkstra's (used in LSR)** – Finds shortest path using nearest unvisited node:

```
Start from source:
  Pick node with smallest distance
  Update distances of its neighbors
  Repeat until all nodes are visited
```

Q3. (C2 – 5 marks) Compare shared memory and **distributed memory** architectures. Provide advantages, disadvantages, and suitable use-cases.

1. Shared Memory Architecture

Aspect	Description
Memory Access	All processors share the same physical memory
Communication	Through **shared variables/memory**
Speed	Fast communication

Aspect	Description
Complexity	Easier programming, harder synchronization

✅Advantages:

- Simple to program and debug
- Faster data access and sharing

❌Disadvantages:

- Hard to scale (limited number of processors)
- Risk of memory conflicts (need locks)

📌Use-cases:

- Multicore processors
- Single-node parallel processing

🌐 2. Distributed Memory Architecture

Aspect	Description
Memory Access	Each processor has its **own private memory**
Communication	Via **message passing (e.g., MPI)**
Speed	Slower communication due to network delay
Complexity	More complex programming

✅Advantages:

- Highly scalable
- No memory conflicts

❌Disadvantages:

- Complex code (need explicit communication)

- Slower communication

📌 *Use-cases:*

- Supercomputers, cloud clusters
- Large-scale data processing (e.g., Hadoop, MPI)

Q4. (C3 – 5 marks) Explain how **barrier synchronization** works in MPI. Provide an example where synchronization is necessary.

◆ What is Barrier Synchronization?

In MPI, **barrier synchronization** means **all processes must stop and wait** at a certain point (the "barrier") until **every process reaches it**. After that, they all move forward **together**.

MPI provides this using:

```
MPI_Barrier(MPI_COMM_WORLD);
```

◆ Why It's Needed? (Example)

Imagine every process is loading data.
If one process starts using the data **before others finish loading**, it can cause errors.

So we use `MPI_Barrier()` to make sure:

- **All processes finish loading data**
- **Then start computing together**

◆ Simple Example:

```
// All processes load data
// Then wait at the barrier
MPI_Barrier(MPI_COMM_WORLD);
// All start computation together
```

Q5. (C2 – 5 marks) Name the **taxonomy** that categorizes computers into four types. Discuss the types briefly.

◆ Name of the Taxonomy:

Flynn's Taxonomy

◆ Four Types of Computers in Flynn's Taxonomy:

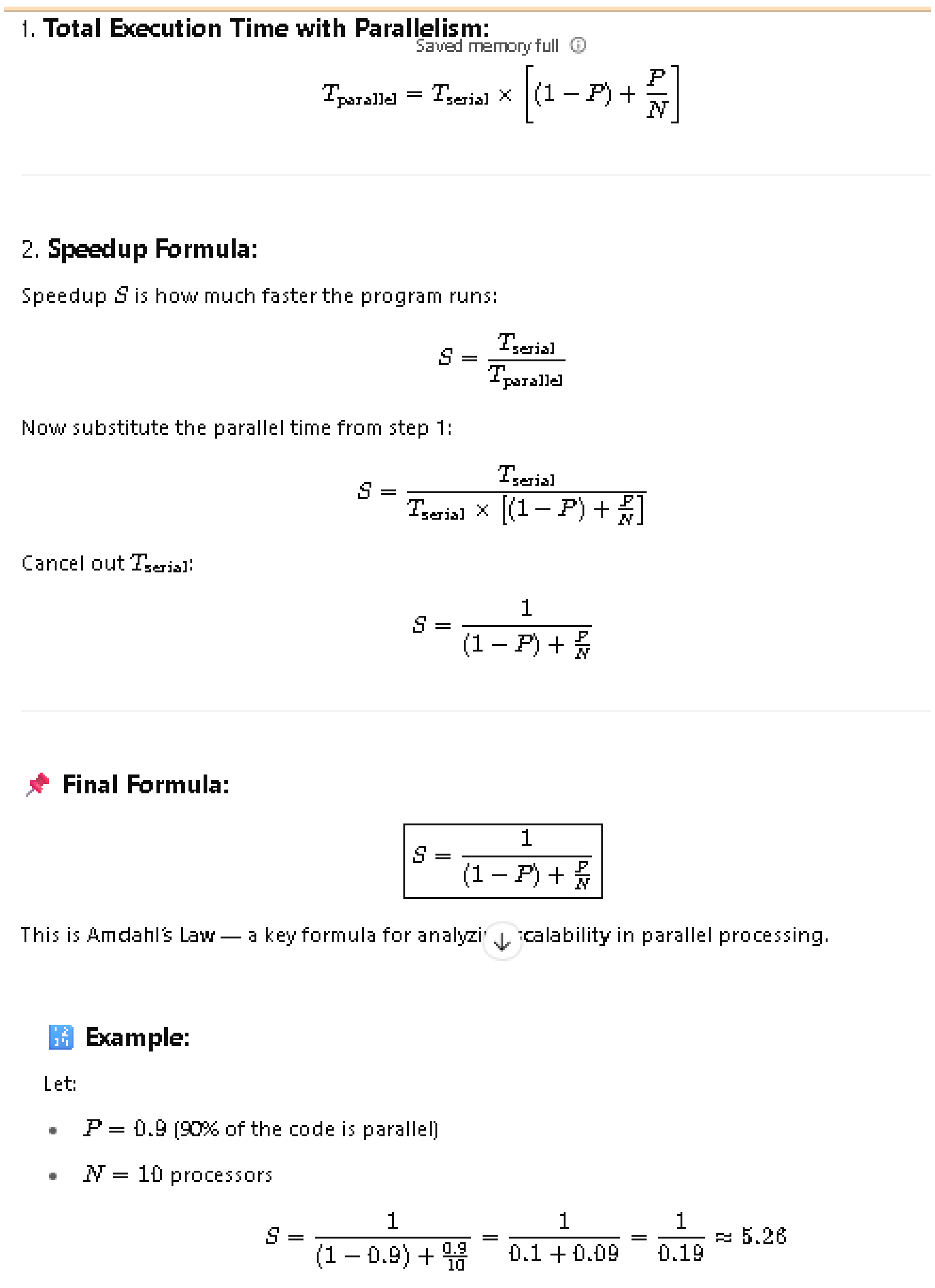
1. **SISD (Single Instruction, Single Data):**
    - One processor executes **one instruction** on **one data** at a time.
    - Example: Traditional single-core computer.
2. **SIMD (Single Instruction, Multiple Data):**
    - One instruction is applied to **multiple data elements** at the same time.
    - Example: Graphics Processing Units (GPUs).
3. **MISD (Multiple Instruction, Single Data):**
    - Many instructions operate on the **same data**.
    - Rare in practice; mostly used for **fault-tolerant systems**.
4. **MIMD (Multiple Instruction, Multiple Data):**
    - Many processors execute **different instructions** on **different data**.
    - Example: Multi-core processors, parallel systems.

Q6. (C2 – 5 marks) Derive the formula for **scalability** that affects speedup in parallel processing and explain with a numeric example.

✓ Step-by-Step Derivation:

Let:

- **T_serial** = time taken by the program with 1 processor
- **P** = fraction of the program that can be parallelized
- **(1 − P)** = serial portion (cannot be parallelized)
- **N** = number of processors



1. **Total Execution Time with Parallelism:**

$$T_{\text{parallel}} = T_{\text{serial}} \times \left[(1-P) + \frac{P}{N}\right]$$

2. **Speedup Formula:**

Speedup S is how much faster the program runs:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

Now substitute the parallel time from step 1:

$$S = \frac{T_{\text{serial}}}{T_{\text{serial}} \times \left[(1-P) + \frac{P}{N}\right]}$$

Cancel out T_{serial}:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

📌 **Final Formula:**

$$\boxed{S = \frac{1}{(1-P) + \frac{P}{N}}}$$

This is Amdahl's Law — a key formula for analyzi calability in parallel processing.

🔢 **Example:**

Let:

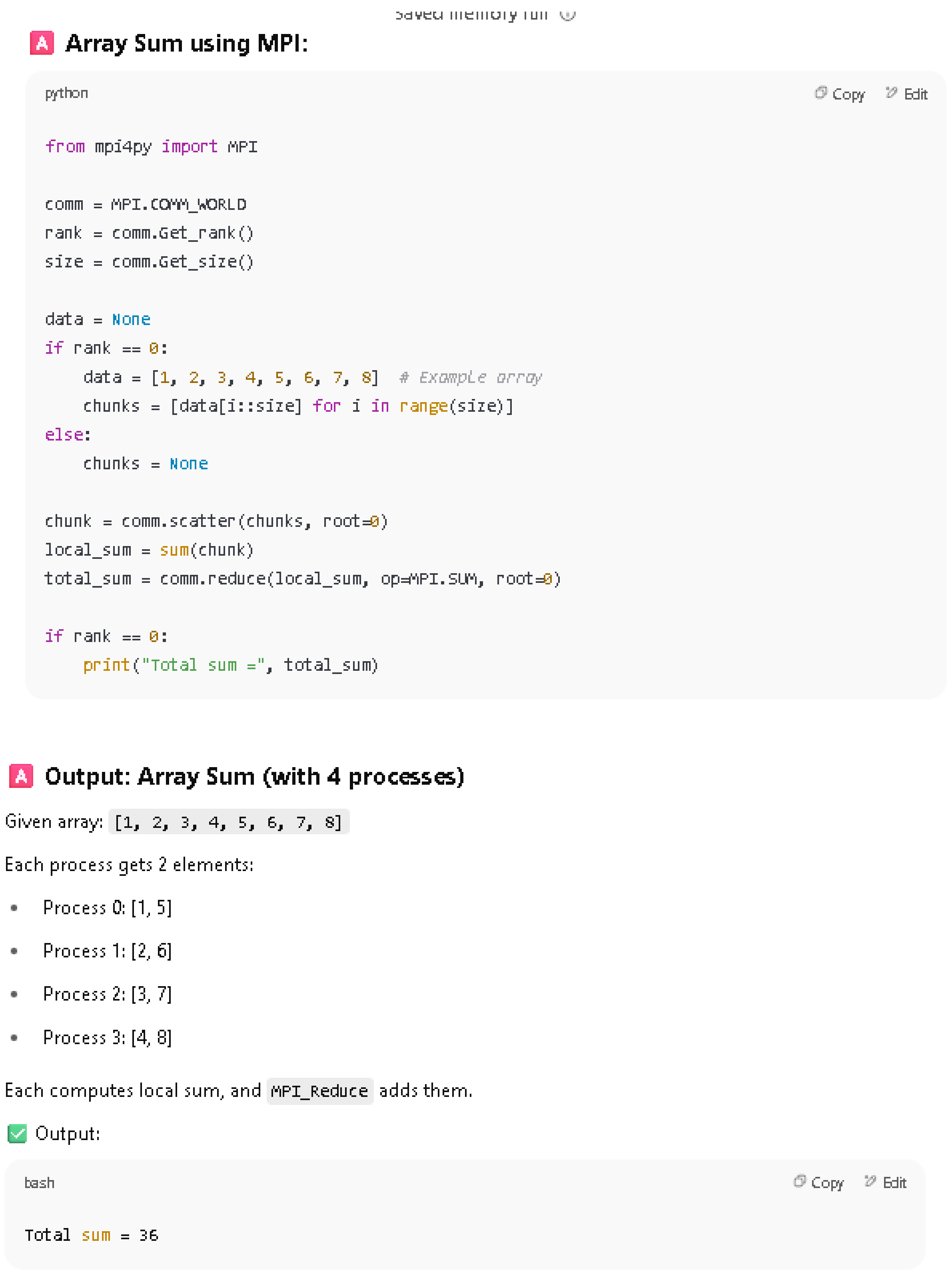
- $P = 0.9$ (90% of the code is parallel)
- $N = 10$ processors

$$S = \frac{1}{(1-0.9) + \frac{0.9}{10}} = \frac{1}{0.1 + 0.09} = \frac{1}{0.19} \approx 5.26$$

Result:

With 10 processors and 90% parallel code, you get around **5.26× speedup**.
Even with more processors, the 10% serial part limits the maximum speedup.

Q7. (C3 – 3 marks) Implement a computer program that uses MPI to calculate the sum of all elements in an array / Matrix multiplication.



Array Sum using MPI:

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

data = None
if rank == 0:
    data = [1, 2, 3, 4, 5, 6, 7, 8]  # Example array
    chunks = [data[i::size] for i in range(size)]
else:
    chunks = None

chunk = comm.scatter(chunks, root=0)
local_sum = sum(chunk)
total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)

if rank == 0:
    print("Total sum =", total_sum)
```

Output: Array Sum (with 4 processes)

Given array: `[1, 2, 3, 4, 5, 6, 7, 8]`

Each process gets 2 elements:

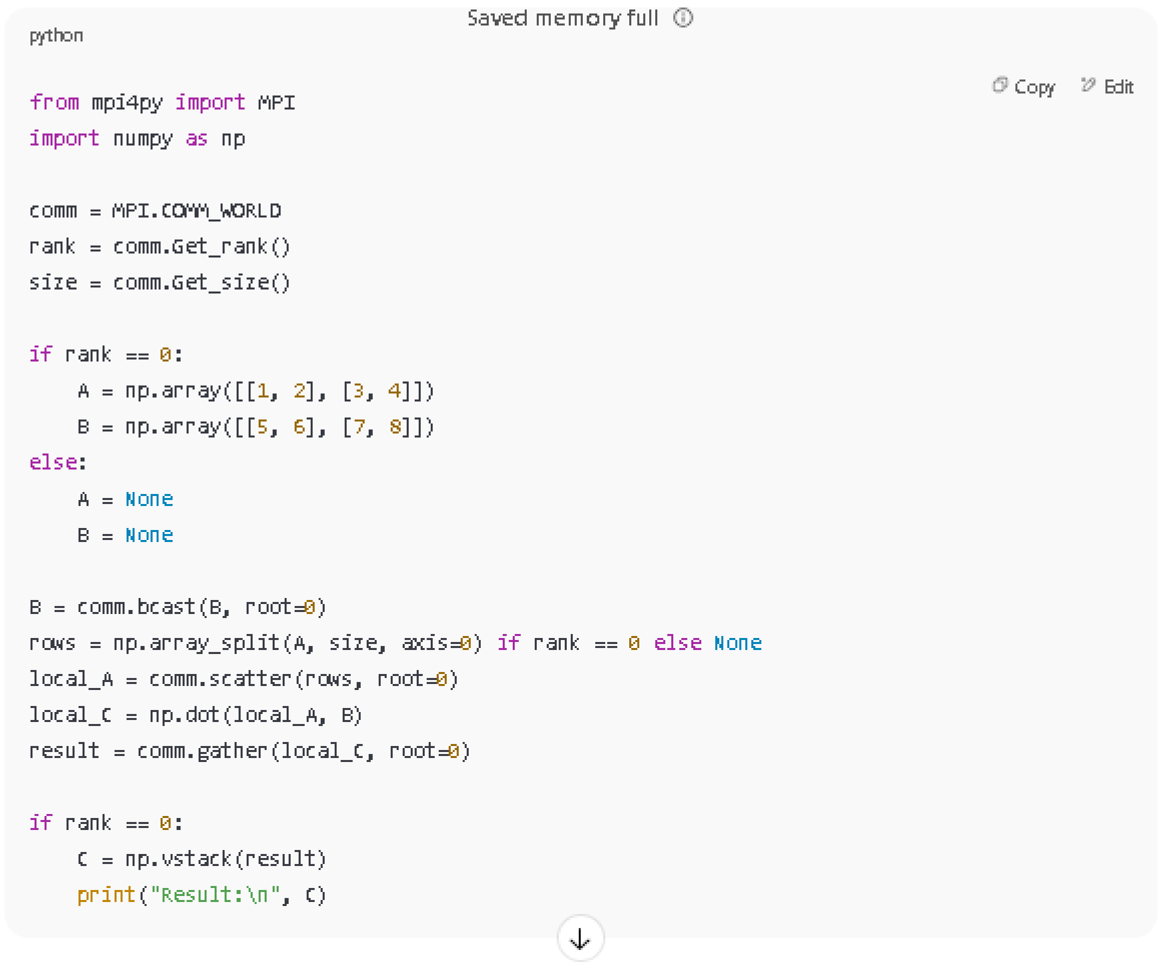
- Process 0: [1, 5]
- Process 1: [2, 6]
- Process 2: [3, 7]
- Process 3: [4, 8]

Each computes local sum, and `MPI_Reduce` adds them.

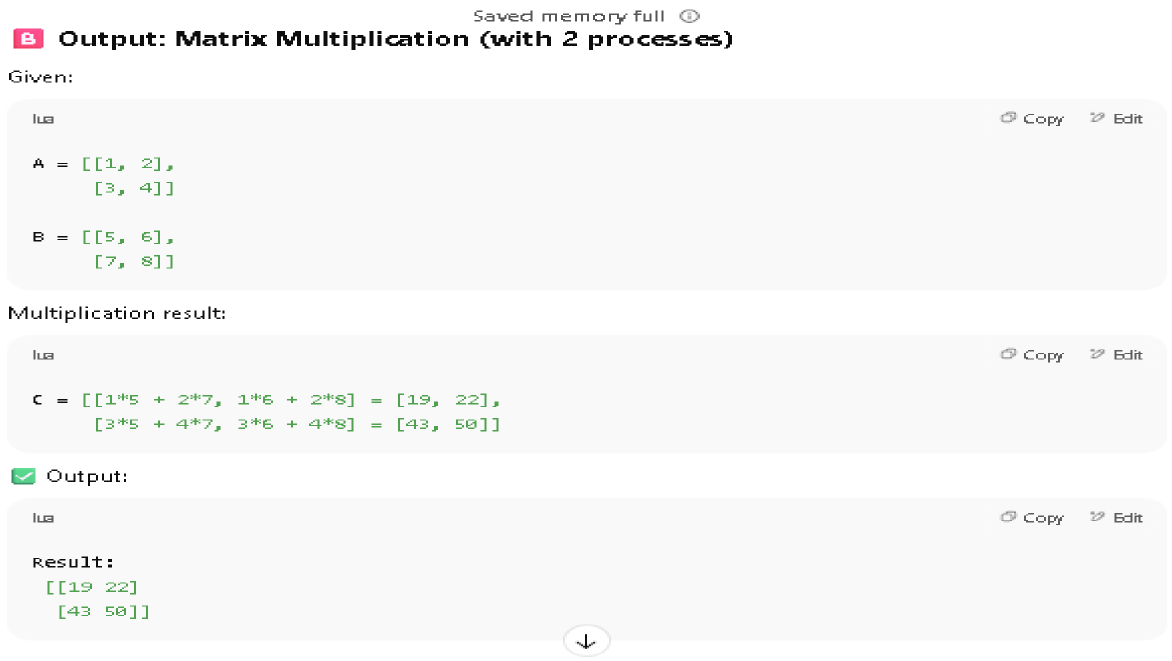
✅ Output:

```bash
Total sum = 36
```

🅱️ Matrix Multiplication using MPI:



```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    A = np.array([[1, 2], [3, 4]])
    B = np.array([[5, 6], [7, 8]])
else:
    A = None
    B = None

B = comm.bcast(B, root=0)
rows = np.array_split(A, size, axis=0) if rank == 0 else None
local_A = comm.scatter(rows, root=0)
local_C = np.dot(local_A, B)
result = comm.gather(local_C, root=0)

if rank == 0:
    C = np.vstack(result)
    print("Result:\n", C)
```



🅱️ Output: Matrix Multiplication (with 2 processes)

Given:

```lua
A = [[1, 2],
     [3, 4]]

B = [[5, 6],
     [7, 8]]
```

Multiplication result:

```lua
C = [[1*5 + 2*7, 1*6 + 2*8] = [19, 22],
     [3*5 + 4*7, 3*6 + 4*8] = [43, 50]]
```

✅ Output:

```lua
Result:
 [[19 22]
  [43 50]]
```

Q8. (C6 – 3 marks) Develop a simple OpenMP program for matrix-vector multiplication.

🔹 **Python Code (Short):**

python Copy Edit

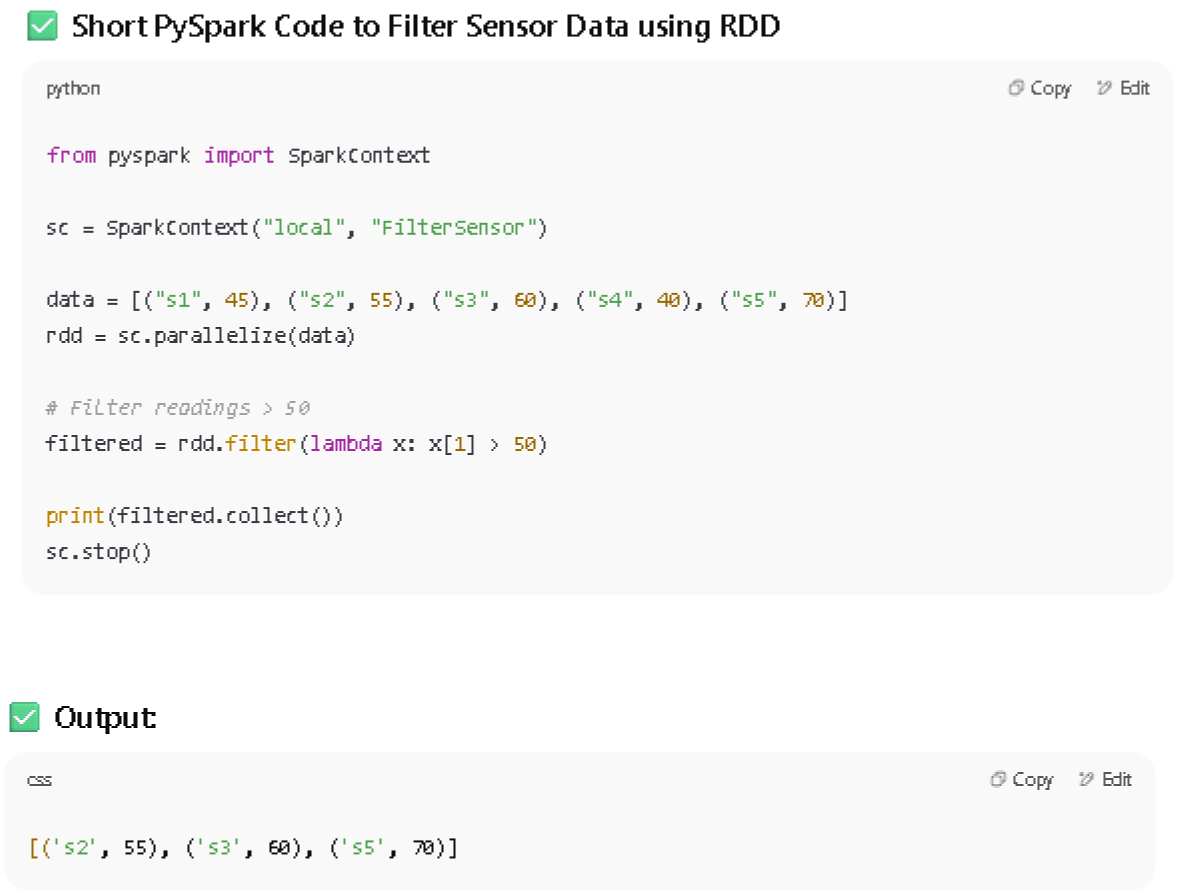
```python
import numpy as np
from joblib import Parallel, delayed

A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
x = np.array([1, 2, 3])

# Parallel multiplication: each row * vector
def multiply_row(row):
    return np.dot(row, x)

y = Parallel(n_jobs=3)(delayed(multiply_row)(row) for row in A)

print("Result vector y:", y)
```

✅ **Output:**

cpp Copy Edit

```
Result vector y: [14, 32, 50]
```

Q9. (C6 – 3 marks) Create an Apache Spark program to filter sensor data using RDD transformations.



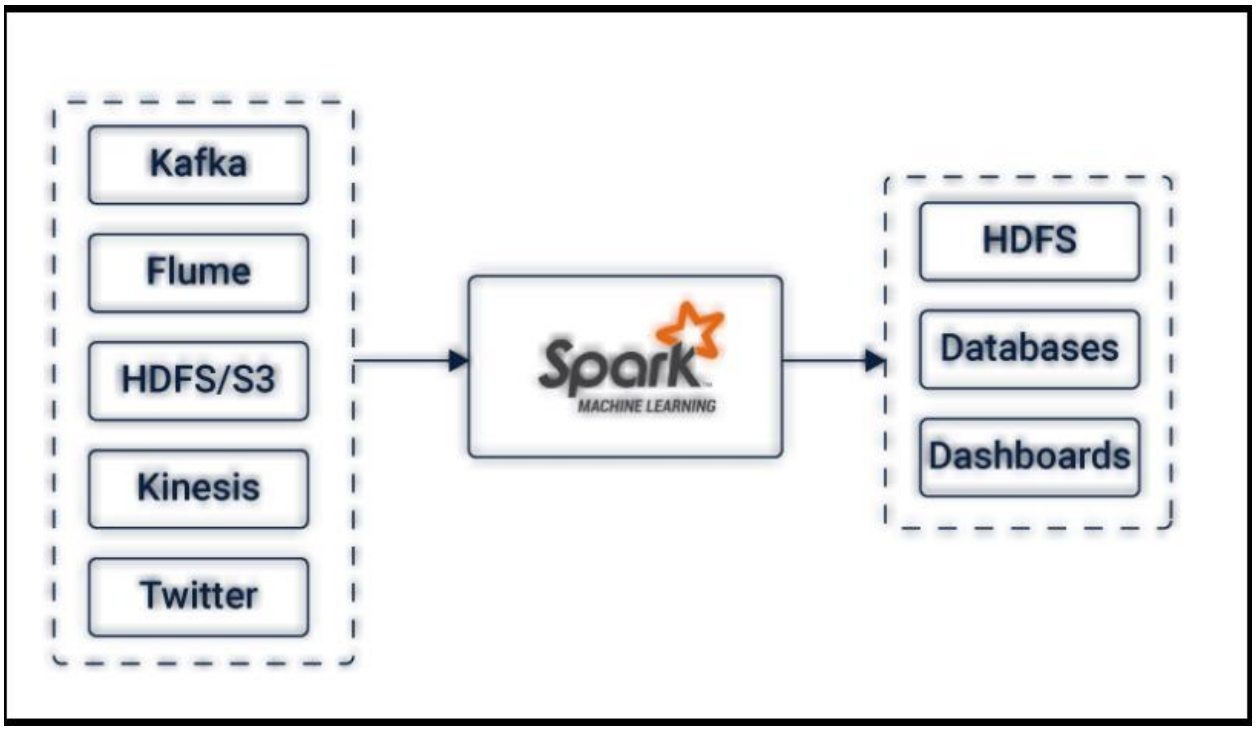
✅ Short PySpark Code to Filter Sensor Data using RDD

```python
from pyspark import SparkContext

sc = SparkContext("local", "FilterSensor")

data = [("s1", 45), ("s2", 55), ("s3", 60), ("s4", 40), ("s5", 70)]
rdd = sc.parallelize(data)

# Filter readings > 50
filtered = rdd.filter(lambda x: x[1] > 50)

print(filtered.collect())
sc.stop()
```

✅ Output:

```css
[('s2', 55), ('s3', 60), ('s5', 70)]
```

Q10. (C6 – 3 marks) Design a real-time data pipeline using Apache Kafka, Spark Streaming, and HDFS.



Q11. (C6 – 3 marks) Design a CUDA kernel to add two arrays in parallel on a GPU.

✅ Python (CUDA) – Add Two Arrays in Parallel (Short)

python Copy Edit

```python
from numba import cuda
import numpy as np

@cuda.jit
def add(a, b, c):
    i = cuda.grid(1)
    if i < a.size:
        c[i] = a[i] + b[i]

a = np.array([1,2,3], dtype=np.int32)
b = np.array([10,20,30], dtype=np.int32)
c = np.zeros_like(a)

add[1, 3](cuda.to_device(a), cuda.to_device(b), cuda.to_device(c))
print(cuda.to_device(c).copy_to_host())
```

✅ Output:

makefile Copy Edit

```makefile
Result: [11 22 33 44 55]
```

Q12. (C2 – 5 marks)

Define the **Client-Server communication model** in distributed systems. **Describe Distributed Client/server Architecture's Terminologies, sockets, RPC, Clusters in Distributed Systems, Multicore, Multiprocessor systems, Superscalar Execution,** with diagrams.

Client-Server Communication Model in Distributed Systems:

In a **client-server model**, the **client** requests services, and the **server** provides them. They communicate over a network. This is the core model for web applications, file sharing, and databases.

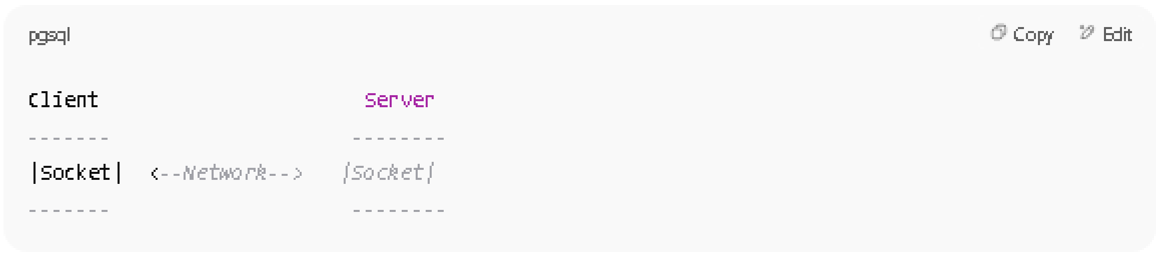
🌐 Distributed Client/Server Architecture Terminologies:

Term	Description
Client	Requests service (e.g., browser, app)
Server	Offers resources or services (e.g., database server)
Middleware	Software that connects clients and servers, manages communication
Protocol	Rules for communication (e.g., HTTP, TCP/IP)

🔌 Sockets:

- **Definition**: An endpoint for sending or receiving data across a network.
- **Usage**: Enables communication between client and server.

Diagram:



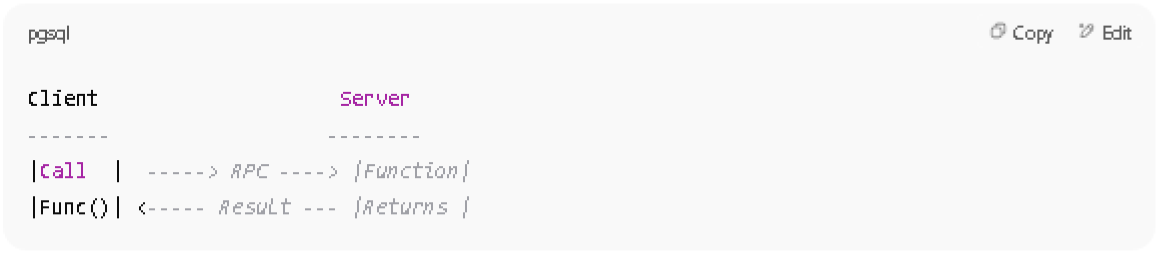
```
Client                    Server
-------                  --------
|Socket|  <--Network-->  |Socket|
-------                  --------
```

RPC (Remote Procedure Call):

- **Definition**: A protocol that lets a program call a procedure on another machine as if it were local.
- **Example**: A login function on a web server being triggered from a client app.

Diagram:

-

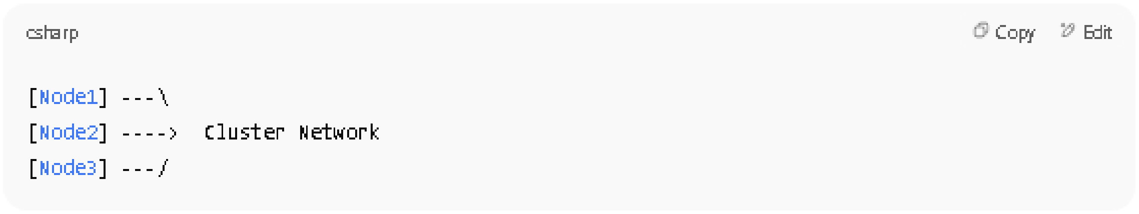


```
Client                    Server
-------                  --------
|Call  |  -----> RPC ----> |Function|
|Func()| <----- Result --- |Returns |
```

Clusters in Distributed Systems:

- **Definition**: A group of interconnected computers (nodes) working together as a single system.
- **Benefit**: High availability and parallel processing.

Diagram:

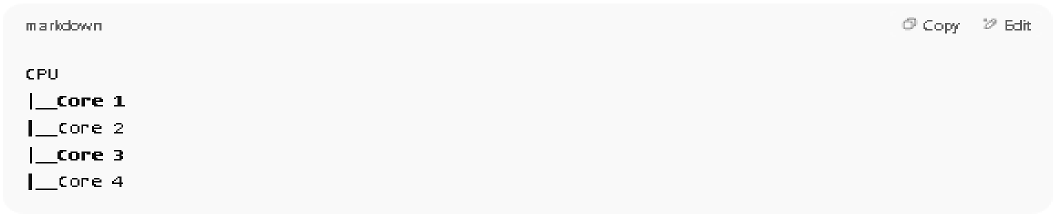


```
[Node1] ---\
[Node2] ---->  Cluster Network
[Node3] ---/
```

Multicore Systems:

- **Definition**: A single processor with multiple cores.
- **Use**: Improves performance by parallel execution.

Diagram:

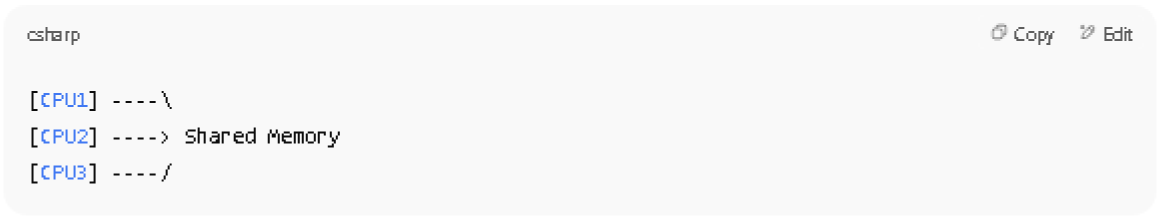


```
CPU
|__Core 1
|__Core 2
|__Core 3
|__Core 4
```

Multiprocessor Systems:

- **Definition**: A system with more than one processor.
- **Example**: Servers with dual or quad CPUs.

Diagram:

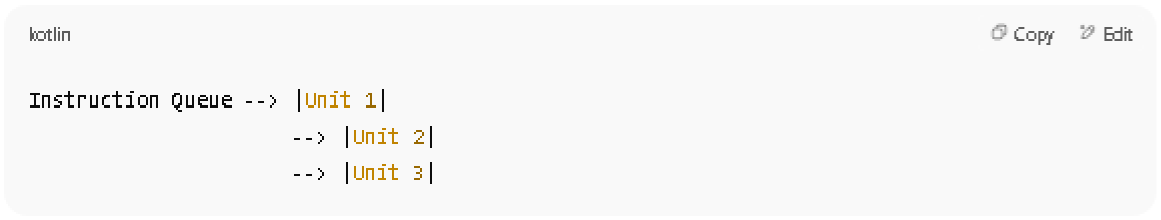


```csharp
[CPU1] ----\
[CPU2] ----> Shared Memory
[CPU3] ----/
```

Superscalar Execution:

- **Definition**: Ability of a processor to execute more than one instruction per clock cycle using multiple execution units.
- **Example**: Modern CPUs fetching and decoding multiple instructions at once.

Diagram:



```kotlin
Instruction Queue --> |Unit 1|
                      --> |Unit 2|
                      --> |Unit 3|
```

Here are the **Quiz 06 Questions Only**:

Q1. (C2 – 5 marks)

Describe the three types of parallelism: data parallelism, task parallelism, and pipeline parallelism. Give real-world examples.

Three Types of Parallelism:

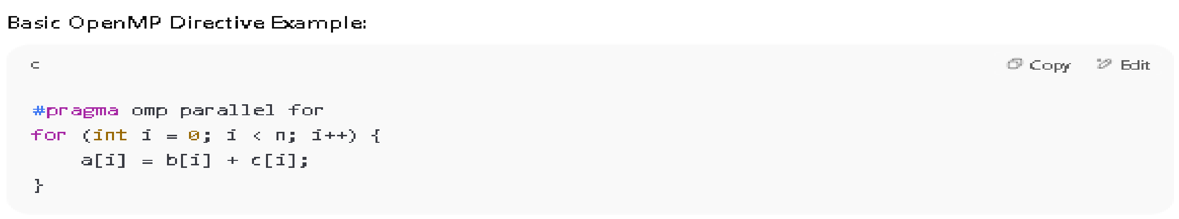
1. **Data Parallelism**
   - *Definition*: Same operation on different chunks of data in parallel.
   - *Example*: Image processing – applying filters to each pixel independently.
2. **Task Parallelism**
   - *Definition*: Different tasks run in parallel, possibly on different data.
   - *Example*: In a web browser, loading a webpage while playing a video.
3. **Pipeline Parallelism**
   - *Definition*: Tasks are divided into stages, each handled by a different processor.
   - *Example*: Assembly line in a factory – each worker does one step in the process.

Q2. (C6 – 5 marks)

Compare OpenMP and CUDA in terms of programming model, hardware requirements, memory access, and applications. Provide a basic OpenMP directive example.

Comparison: OpenMP vs CUDA

Feature	OpenMP	CUDA
Programming Model	Shared-memory, thread-based	SIMT (Single Instruction, Multiple Thread)
Hardware	Multicore CPUs	NVIDIA GPUs
Memory Access	Shared among threads	Separate host and device memory
Applications	Scientific computing, simulations	Deep learning, graphics, matrix ops



Basic OpenMP Directive Example:

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

Q3. (C3 – 5 marks)

Using MPI, write a C-style code snippet to calculate the sum of an array using message-passing across 4 processes. Comment on synchronization and data distribution.

Same As Q7

Q4. (C6 – 5 marks)

Develop a small-scale cloud deployment plan using Hadoop ecosystem components (HDFS, YARN, MapReduce) for a university's log management system.

Cloud Deployment Plan for University Log Management using Hadoop Ecosystem:

1. **HDFS (Hadoop Distributed File System):**
   - Stores log files across distributed nodes.
   - Fault-tolerant and scalable.
2. **YARN (Yet Another Resource Negotiator):**
   - Manages resources and schedules tasks for log processing jobs.
   - Runs MapReduce jobs efficiently on the cluster.
3. **MapReduce:**
   - Parses logs for analysis (e.g., error frequency, usage stats).
   - Mapper: filters useful info (timestamps, errors).
   - Reducer: aggregates and summarizes results.

Setup Overview:

- Logs from servers stored in HDFS.
- Scheduled MapReduce jobs (via YARN) analyze and report patterns.
- Results stored in HDFS or exported for dashboards.