APPROACHES TO PARALLEL PROGRAMMING

When it comes to parallel programming, the main goal is to split up tasks so that multiple processors can work on different parts at the same time. There are several ways to approach this, and each method suits different kinds of problems.

a. Data Parallelism

This is one of the most common approaches:

- In data parallelism, the data is divided into chunks, and each processor works on a different chunk of the data. However, they all do the same task on their respective chunk. For example, imagine you need to multiply two large matrices. You can split the rows of these matrices into chunks and give each chunk to a different processor. Every processor performs the same multiplication operation, just on different parts of the data.
- Best for Large datasets where the same operation needs to be applied repeatedly.

b. Process Parallelism

This approach focuses on splitting tasks (or processes) instead of data:

- Each processor handles a different task or function. For example, in a video game, one processor might handle graphics rendering, another might handle sound, and another handles the game logic. Here, the different tasks don't need to be identical, and each processor can work on a different part of the overall job.
- Best for when a job can be broken down into several distinct tasks, each of which can run independently.

c. Farmer-Worker Model (or Master-Slave Model)

This model is inspired by the way a manager (farmer) might distribute tasks to workers:

- In this model, there's one main processor called the "master" (or farmer). The master assigns smaller tasks to other processors, called "workers." The workers do their assigned

task and send the results back to the master. The master then compiles the results and either sends more tasks or completes the job. This is like a factory, where one boss directs the workers, and the workers return finished products.

- Best for when the main task can be split into independent subtasks, and there's a central controller (master) that can manage the workflow efficiently.

d. Levels of Parallelism

Parallelism can occur at different levels, depending on how big the chunks are:

- **Large Parallelism:** Whole processes can be parallelized. Think of running multiple independent programs at the same time.
- **Medium Parallelism:** Functions or methods within a program are parallelized.
- **Fine Parallelism:** Small pieces of code, like loops, are parallelized.

- **Very Fine Parallelism:** Individual instructions or blocks of instructions are parallelized, often managed by the compiler or processor itself.

e. Challenges of Parallel Programming

While parallel programming can greatly speed up tasks, it also introduces challenges:

- **Task Dependency:** Some tasks can't be done until others are finished, which can create delays.
- **Synchronization:** When multiple processors are working on related tasks, they may need to communicate or share data. Ensuring this happens without causing slowdowns is tricky.
- **Load Balancing:** It's important to make sure that each processor gets a fair amount of work. If one processor has too much to do while others are idle, the system's efficiency drops.

INTER-PROCESS COMMUNICATION (IPC)

IPC is the way different processes in a computer system communicate and share data with each other. In a distributed or parallel system, processes are often spread across different computers or processors, so they need a method to exchange information.

a. What is IPC?

- **Processes:** A process is a running instance of a program. Sometimes, you need multiple processes running at the same time to solve different parts of a problem. But if these processes need to work together, they must communicate.
- IPC allows these processes to exchange data, send messages, or synchronize their actions.

b. Why is IPC Important?

Without IPC, processes running on different parts of a system wouldn't be able to collaborate or share their results. IPC is essential in systems where tasks are divided among multiple processes, such as in parallel computing or distributed systems (like cloud computing).

c. Types of IPC

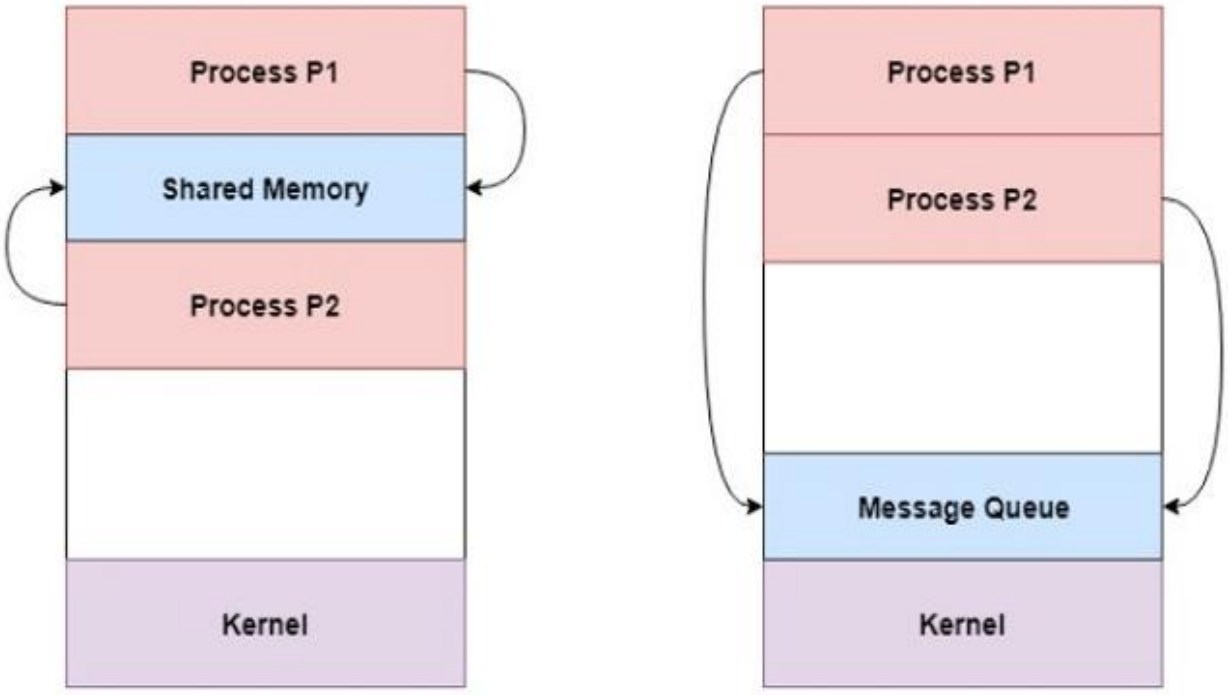
There are different methods for IPC, depending on the system's architecture:

- **Shared Memory:** In this method, multiple processes can access the same area of memory. This is common in parallel systems where processes share the same physical machine. It's fast because no data needs to be sent over a network, but it can be complicated because processes need to be careful not to overwrite each other's data.
- **Message Passing:** In distributed systems, processes don't share memory. Instead, they communicate by sending messages to each other. Each process sends a message (containing data or instructions), and the receiving process

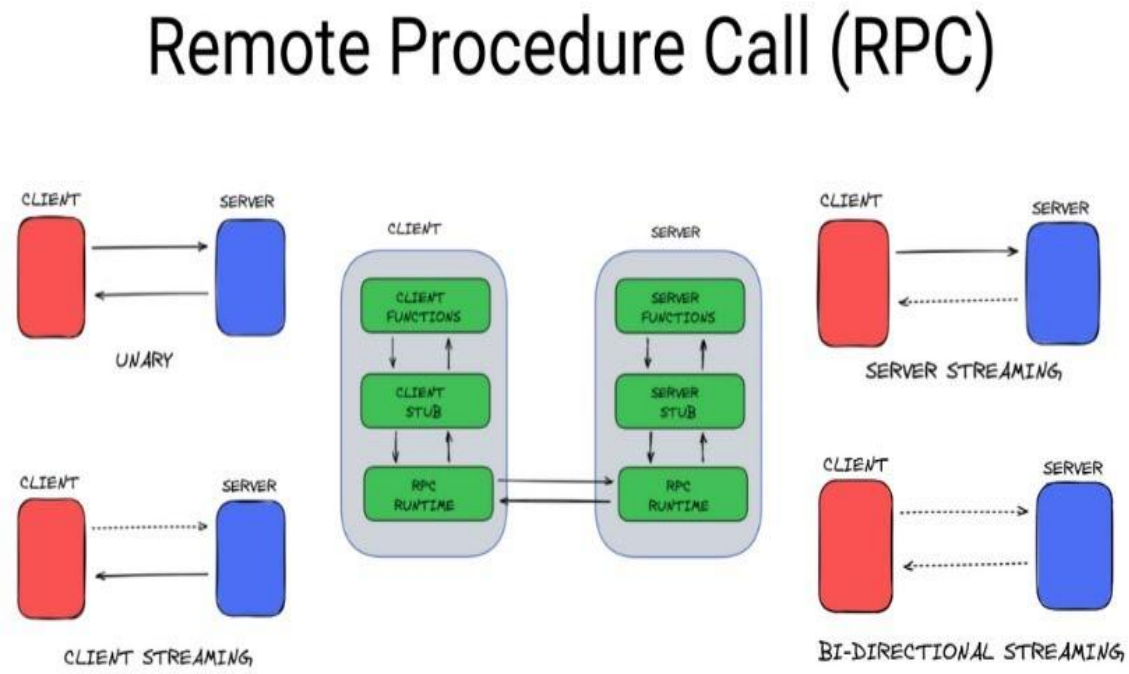
acts on it. This is slower than shared memory but is essential when processes are spread out over different computers.

- **Pipes and Queues:** Pipes allow two processes to communicate by sending data through a "pipe" (one process writes, and the other reads). Queues allow multiple processes to put messages in a shared line (queue) that others can read from.

Approaches to Interprocess Communication



REMOTE PROCEDURE CALL (RPC)



RPC is a powerful tool in distributed computing. It allows a program to call a function (or procedure) on another computer as if it were calling a local function.

a. What is RPC?

- **Local Function Call:** Normally, when a program runs, it calls functions or methods that exist on the same machine.
- **Remote Function Call:** RPC allows a program to call a function on another machine across a network. The program calling the function doesn't need to know where the function

is physically located; it just makes the call as if the function were local.

b. How Does RPC Work?

1. **Client and Server:** There's a client (the program that makes the call) and a server (the program that hosts the function).
2. **Call Request:** The client sends a request to the server saying, "Run this function with these inputs."
3. **Server Execution:** The server receives the request, runs the function, and then sends the result back to the client.
4. **Result:** The client receives the result as if the function had been executed locally.

c. Benefits of RPC

- **Simplicity:** RPC makes distributed computing easier because developers can write programs as if they are running on a single machine, even though parts of the program are executed on different machines.
- **Abstraction:** The client program doesn't need to worry about where or how the function is executed—it just makes the call.

d. Challenges of RPC

- **Latency:** Since the function call is happening over a network, it's much slower than a local function call.
- **Failure Handling:** If the server crashes or the network fails, the client might not receive a response, so additional error handling is required.
- **Security:** Exposing functions across a network introduces security concerns, as malicious actors could try to exploit them.