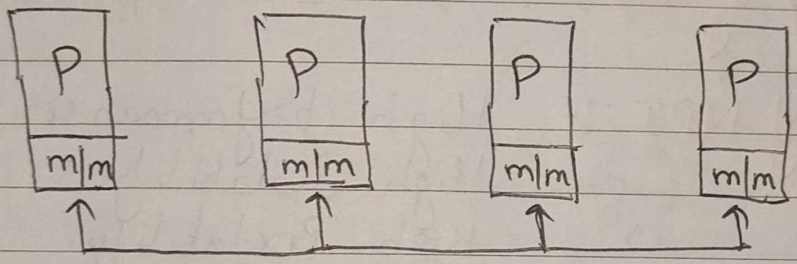
HPC

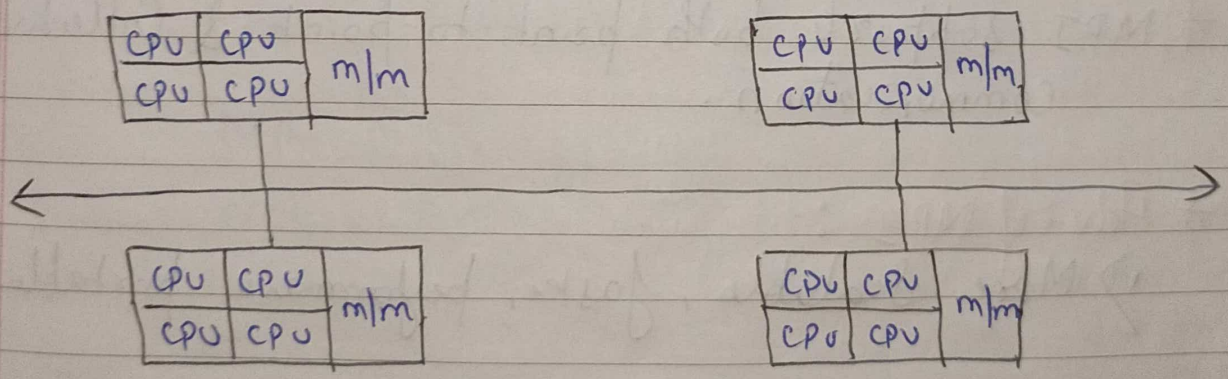
Unit-5

MPI :- (Message Passing Interface)

→ MPI concept came in 1990s.
→ It is used for communication between two or more processors.

* MPI is a standardise means of exchanging message between multiple computers running parallely across distributed memory.





Note :- MPI is needed when there is connection between distributed memory system. We don't need for shared memory system.
(i.e required when there is no way for one processor to directly access the address of another).
However, it can be regarded as programming model

and used on shared-memory or hybrid system.

Need of MPI? → It provides mechanism to create processors to execute process on different processors. Also provide mechanism to send and receive message.

* In earlier days, we were using MPMD model which had certain limitations but nowadays we use SPMD which have no restrictions. MPMD uses PVM (Parallel virtual Machine) as library.

* Library of MPI :- #include <mpi.h>.

* Goals of MPI :- High performance
  High Scalability
  High Portability.

* MPI support both point-to-point & Collective communication.

* Adv. of MPI :-
  1) More Scalable, faster, performance, portable

History :-

MPI1 (version 1) → 1994
→ most application runs on this
→ used to create parallel SPMD code
→ available on almost all language C, C++, fortran.

MPI 2 (Version 2) → 1996
→ MPI + One-sided communication
dynamic process Control
Parallel I/O (MPI-I/O)
↑ can change according to nature of model.

MPI with a Code :-

```
#include <stdio.h>
#include <mpi.h>
int main() {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello World, I am %d of %d", rank, size);

    MPI_Finalize();
    return 0;
}
```

→ MPI_Init → Initialize parallel environment
→ MPI_COMM_WORLD → Default Communicator of MPI
→ MPI_COMM_SIZE → No. of processes
→ MPI_COMM_RANK → Rank of calling process.
↓
(unique identifier which identify each process).

→ MPI_Finalize → End parallel environment

Communication → point-to-point
Communication → Collective

Point-to-Point → two processes in the same communicator are going to communicate each other.

Collective → All processes communicate each other.

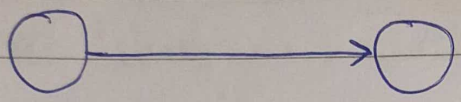


fig. point-to-point

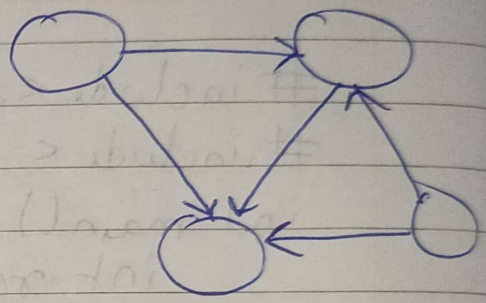
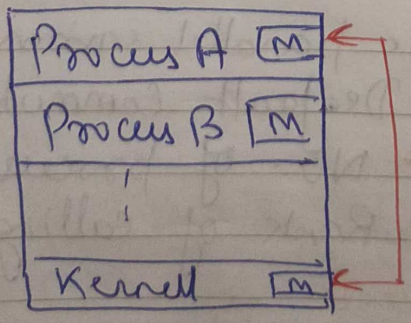


fig. Collective

* We need 2 operations to perform MP. In distributed environment, computer processes may reside in different computers connected by a network. This kind of scenario is called as MP System (MPS). There is no need of shared m/m.



* MPI provides 2 type of operations :-
(a) MPI_Send() (b) MPI_receive()

* ~~Sendrd~~ * Sending and receiving are two fundamental concept of MPI. Almost every single function in MPI can be implemented with basic send & receive calls.

* Message - send by a process can be either fixed or variable.

fixed → The system level implimentation is straight forward as compared to variable. But it creat problem and makes the task of programmer more difficult.

Variable → require more complex system but the programe task become easy or simpler.

Prototype of MPI_Send()	Prototype of MPI_recive()
MPI_Send(void * data, int count, MPI_Datatype datatype int destination, int tag, MPI_Comm Communicator)	MPI_Recv(void * data, int count MPI_Datatype datatype, int source, int tag, MPI_Comm Communicator, MPI_Status * status)

* For communication between A → B, a communication link must exist between two process. This link can be implemented in variety of ways :-

① Direct or Indirect
② Synchronous & Asynchronous
③ Automatic or explicit buffering.

Direct → Each process that want to communicate must explicitly name the communication.

Send(P, message) receive(q, message)

Indirect → not explicitly called.
→ use mailbox

Send a message to mailbox A
receive a message from mailbox A.

Synchronous → Communicate after scheduling.
Ex → Video call
→ require immediate action.

Asynchronous → There is no need of scheduling.
→ No immediate action is required.

Automatic → automatically allocate space
→ provide Queue with indefinite block
→ no need to wait

Explicit buffering → need to wait until the block becomes available
→ Sender cannot send any message till then.

Communication → Blocking
Communication → Non-Blocking

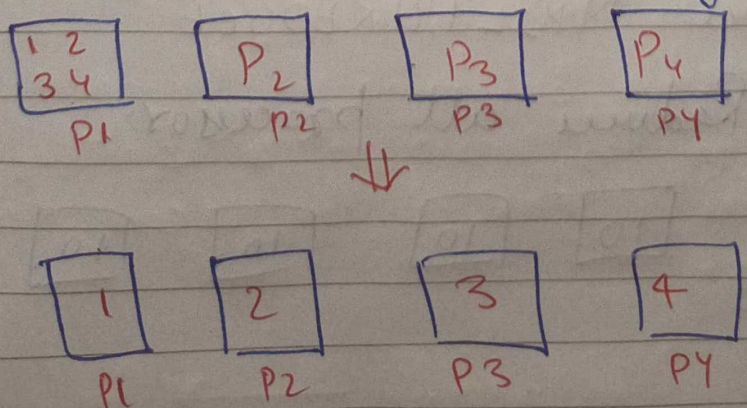
Blocking :- Simultaneous send & receive
function → MPI_SEND(), MPI_RECV()

Non-Blocking :- Immediate execution.
Use queue to store msg.
function → MPI_ISEND(), MPI_IRECV().

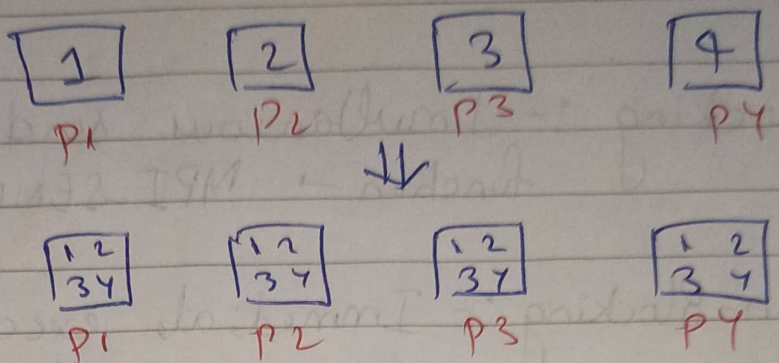
Collective Communication :-
→ One to many
3 types :-

1) Barrier Synchronization :- MPI_BARRIER()
↳ Block until all process have reached a Synchronization point.

2) Data Movement (Global Communication)
↳ Broadcast MPI_BCAST → (to all)
↳ Scatter MPI_SCATTER → Send data from 1 task to all other task in a group.

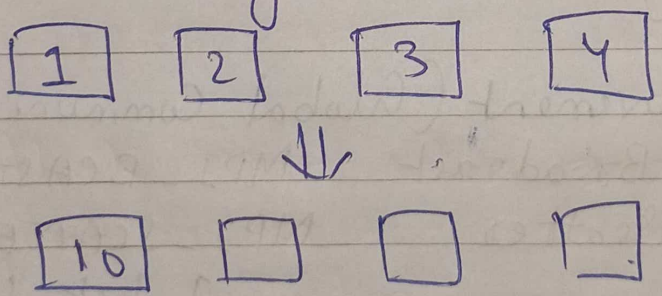


→ Gather : MPI_GATHER → (Opposite to Scatter).
→ All_gather : MPI_ALLGATHER



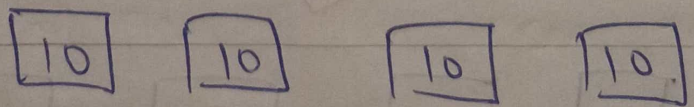
Collective operation (Global Reduction) :-
→ One process from Communicator collects data from each process & performs an operation on that data to compute a result.

* Reduce → MPI_REDUCE
↳ It reduces the value from all process to a single value

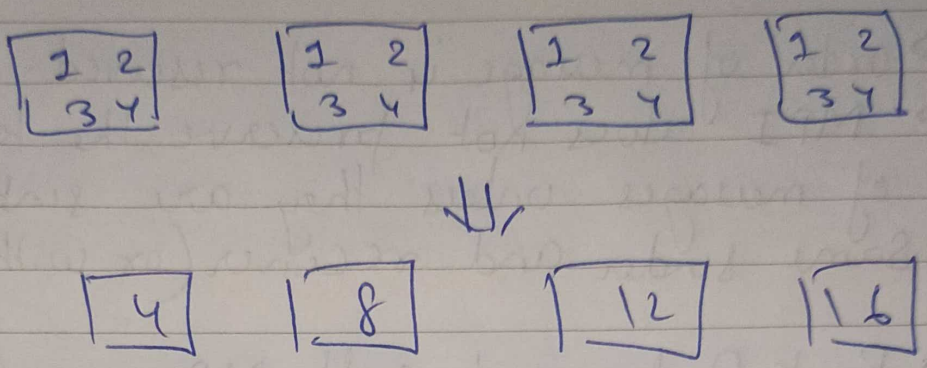


← operation MPI_SUM.

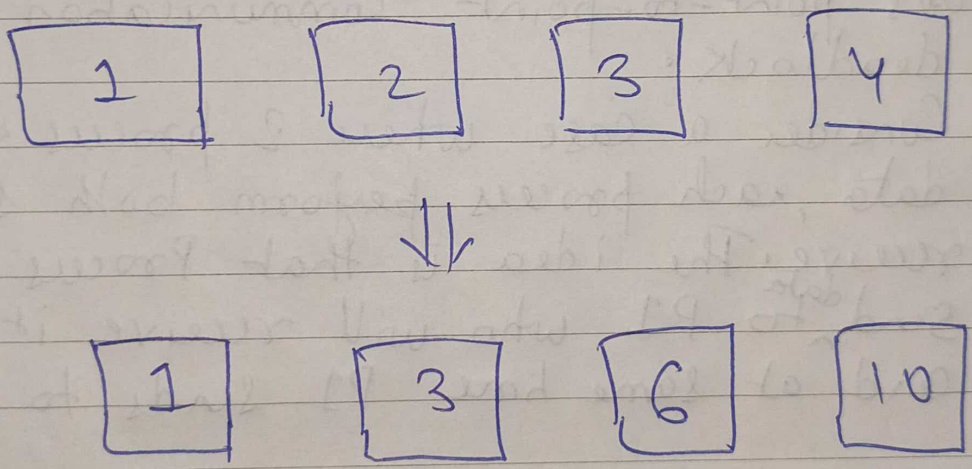
* All Reduce MPI_ALLREDUCE
→ Reduces all processor.



* Reduce Scatter MPI_REDUCE_SCATTER.



MPI Scan :-
Commulative result



Virtual topologies

A virtual topology represent a way that MPI process communicate.

MPI provides 2 types of topologies :-
(i) Cartesian grid based topology
(ii) Graphs Based topology.

(i) Cartesian grid based topology :-
It is a regular grid like structure that can be visualized as 1D, 2D and 3D array of process. To creat this we use mpi_cart_creat() function.
Grid structure are defined by a number of dimensions and no. of process in each coordinat direction. It is mapped using a row-major numbering system. It is machine dependent.

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3
(1,0) 4	(1,1) 5	(1,2) 6	(1,3) 7
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

function :- 1) MPI_Cart_creat (MPI_Comm_ comm-old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm-cart).
2) int MPI_Dims_creat (int nnodes, int ndims, int *dims)

2) Graph based topology :-
→ It can only be used in Intra communicators
→ No. of graph nodes must not be greater than No. of processes.
→ It is a more general structure that can represent arbitrary connections between process.
→ It uses Nodes and edges to represent the process and connection link respectively.

→ MPI_Graph_Create() function is used to creat this.

functions :- 1) MPI_Graph_Creat(MPI_Comm comm_old, int nnodes, int *index, int * edges, int reorder, MPI_Comm *comm_graph).

2) MPI_Topo_test(MPI_Comm Comm, int *top_type)
↳ determine type of topology associated.

3) MPI_Cartdim_get(MPI_Comm comm, int *n dims)
↳ retrieve Cartesian topology information.

4) MPI_Cart_Rank(MPI_Comm Comm, int *coords, int *rank)
↳ determine process rank in communicator given Cartesian location.

5) MPI_Graph_neighbour_Count()
↳ return count of neighbours

6) MPI_Graph_neighbours()
↳ return neighbours

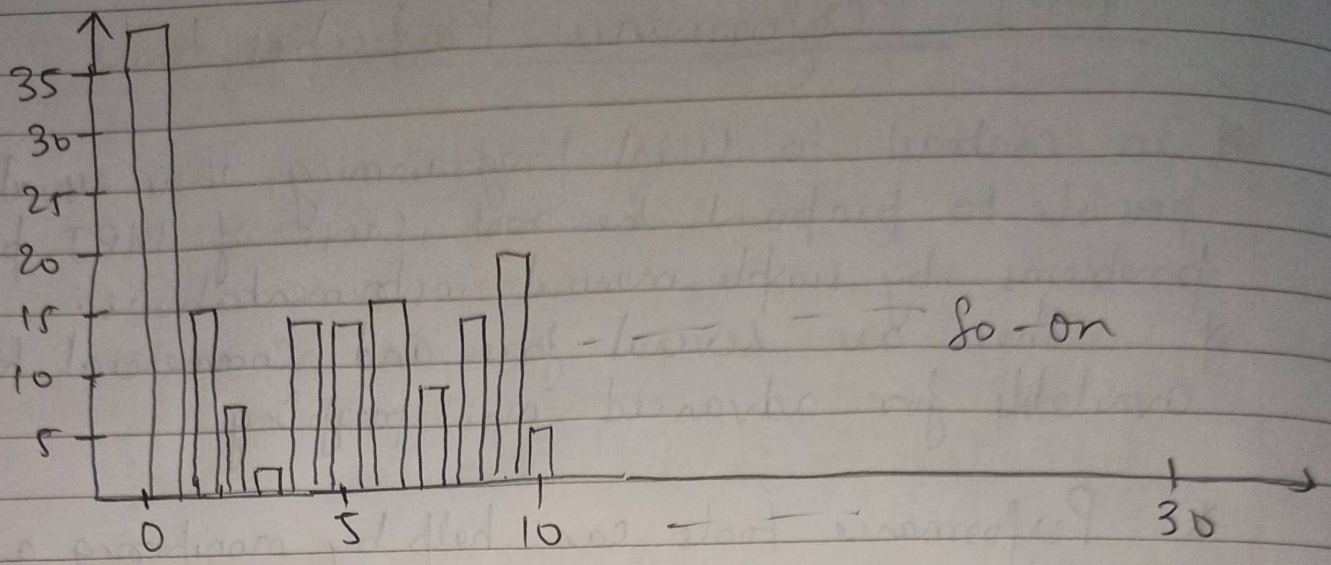
7) MPI_Cart_shift()
↳ for displacement.

MPI Performance ~~Properties~~ tools

* In contrast to serial programming, it is usually not possible to pinpoint the root cause of MPI performance problems by simple manual instrumentations.
* There are several free and commercial tools available for advanced mpi profiling

* Performance tools can help by monitoring a program execution and reducing performance data that can be analysed to locate and understand the area of poor performance.

* IPM is a simple and low-overhead tool that is able to retrieve this information.
* IPM is a portable profile infrastructure for parallel codes. It provides a low overhead performance. It is scalable and easy to use.
* Like most MPI profilers, IPM uses the MPI profiling interface, which is part of standard.
* Each MPI function is wrapped around the actual function whose name starts with 'PMPI_'
* In case of IPM, it is sufficient to preload a dynamic library and run the application. Information about ~~dynamic~~ data volumes (per process and per process pair), time spent in MPI calls, load imbalance etc is then accumulated over the application's runtime and can be viewed in graphical form.



↳ IPM "Communication balance" of a master work style parallel application.

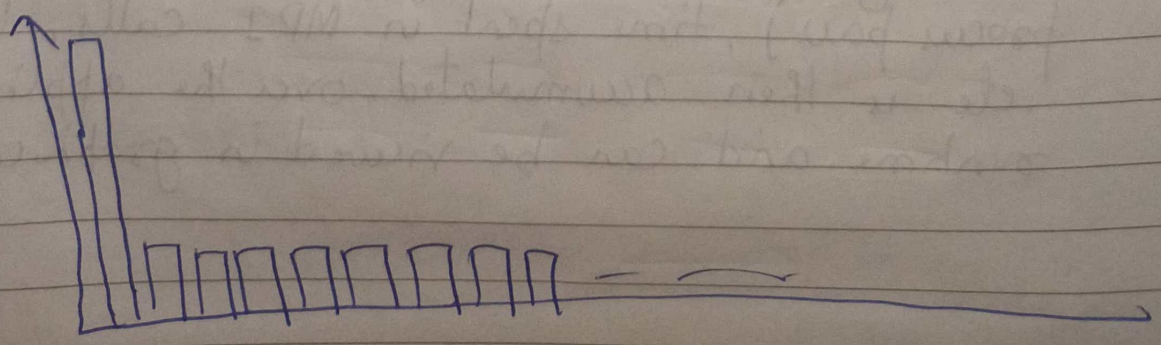
Each bar represent MPI rank and shows how much time the process spend in different MPI functions.

In this, the runtime was 38 seconds.

Rank 0 (the master) distributes the work among the workers, so it spend most of its runtime in MPI_recv() waiting for result.

The workers are obviously quite load imbalanced and between 5 and 50% of their time is wasted waiting at barrier.

A small change in parameter (reducing the size of work pakages) was able to correct this problem. Overall 25% reduction in runtime is seen.



Problems in MPI :-

→ order of message is not reserved.
→ MPI does not preserve the temporal order of messages unless they are sent between same sender and receiver (or with same tag).

→ Rank 0 does not call MPI_rcv() before returning from its own execution

→ A common source of error in program that uses point-to-point communication is called deadlock.
→ Consider a case where 2 process exchange data, each process perform both send and receive. The idea is that Process 0 will send data to P1 who will receive it from P0 and at same time P1 sends to P0.

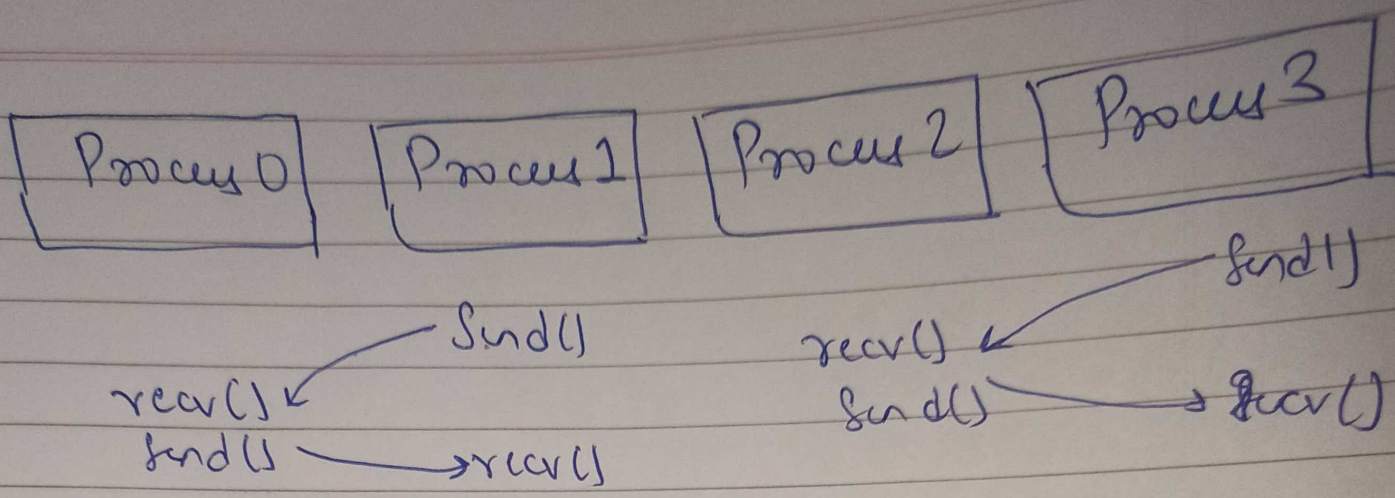
→ Deadlock occur when every process is forced to wait on an action of another process to continue executing.

How to overcome it?

1) One possible solution of fixing a deadlock problem or to avoid it is simply reversing the order of one of the receive or send pairs.

send/receive ⟶ Receive/send

2)



→ odd process are send/recv() pair
→ even process are recv()/send() pair

HPC UNIT – 5

Non-blocking vs asynchronous communication in MPI

In the context of MPI (Message Passing Interface), the terms "non-blocking" and "asynchronous" are often used interchangeably, but they have slightly different meanings. Let's explore the difference between the two:

1. Non-blocking communication: Non-blocking communication refers to the ability to initiate a communication operation and continue executing other tasks without waiting for the operation to complete. In MPI, non-blocking communication is achieved using functions like `MPI_Isend` and `MPI_Irecv`. When you use these non-blocking communication functions, the MPI library returns immediately, allowing your program to perform other computations while the communication operation progresses in the background. However, you still need to ensure that you don't access the data involved in the communication until the operation is completed, to avoid race conditions.

2. Asynchronous communication: Asynchronous communication, in the context of MPI, implies that multiple communication operations can be in progress simultaneously, and they can progress independently of each other. In other words, you can initiate multiple communication operations and not necessarily wait for each of them to complete before initiating others. This level of asynchrony allows for potential overlap of communication and computation, leading to increased performance. Non-blocking communication is a key technique used to achieve asynchronous communication in MPI.

In summary, non-blocking communication is a mechanism that allows you to initiate a communication operation and continue with other tasks, while asynchronous communication is a broader concept that encompasses the ability to have multiple communication operations progressing simultaneously. Non-blocking communication is often used to enable asynchronous communication in MPI, but it's important to note that not all non-blocking communication is necessarily asynchronous if only one communication operation is being used.

Explain 1. serialization 2. Synchronization 3. Contention 4. Implicit serialization in MPI.

1. Serialization: Serialization in MPI refers to the process of converting data from a structured format in memory to a linear format suitable for transmission over the network or storage. When sending or receiving data using MPI communication routines, the data needs to be serialized before transmission and deserialized upon reception. Serialization ensures that the data can be transmitted as a sequence of bytes, independent of the memory layout or data structures used by the sender and receiver processes. MPI provides functions like `MPI_Pack` and `MPI_Unpack` to facilitate serialization and deserialization of complex data types.

2. Synchronization: Synchronization in MPI involves coordinating the execution of multiple processes to ensure that they reach specific points in their execution in a coordinated manner. Synchronization is essential to enforce dependencies, order of operations, and consistency in parallel programs. In MPI, synchronization can be achieved using various mechanisms, such as collective communication operations like `MPI_Barrier`, where processes wait until all processes in a communicator reach the barrier before continuing execution. Synchronization is crucial in parallel algorithms to avoid race conditions and maintain correctness.

3. Contention: Contention refers to a situation in which multiple processes or threads are competing for the same resource or piece of data, leading to potential conflicts and decreased performance. In the context of MPI, contention can arise when multiple processes simultaneously access shared resources, such as memory locations or I/O devices, leading to contention for access to those resources. Contention can result in increased communication delays, decreased scalability, and even deadlocks. Proper design and synchronization techniques are necessary to minimize contention and ensure efficient execution in parallel programs.

4. Implicit serialization: Implicit serialization in MPI refers to the automatic handling of data serialization and deserialization by MPI library functions. When using point-to-point communication routines like `MPI_Send` and `MPI_Recv`, MPI automatically serializes the data being sent and deserializes it upon reception, abstracting away the serialization process from the programmer. This implicit serialization simplifies the programming model and allows the developer to focus on the application logic rather than low-level data serialization details. However, it's important to note that implicit serialization might not be as efficient as explicit serialization using `MPI_Pack` and `MPI_Unpack` for certain complex data types, where explicit control over the serialization process may be required for performance optimization or compatibility with external systems.