1. The MPI_WORLD:

   In MPI, the group of all nodes used in a particular MPI application is "world," represented by the variable MPI_COMM_WORLD. This variable is a communicator that provides all the information necessary to do message-passing in the group.

   Function: A communicator constrains the message-passing to remain within the group and prevents nodes outside the group from receiving spurious messages.

Note>:Each node in the group is identified by a unique "rank" that is an integer value.Ranks are numbered 0, 1, 2, ..., n-1 if the total number of nodes in the group is n.

**How to Start and End application:**

To start up an MPI application, all MPI programs must contain one and only one call to MPI_Init, which enables communications.

Syn: MPI_Init(&argc,&argv)

Before exiting the program MPI_Finalize() must be called. The User must check that all the pending communication of the processes must be completed.

**Rank & Size:**

An MPI call MPI_Comm_rank is desired to obtain the process's rank. It is basically a process identifier.
Syn: MPI_Comm_rank(MPI_Comm_world, &rank);

To find out the size of the communication world one can call MPI_Comm_size.
Syn: MPI_Comm_size(MPI_Comm_world, &size);

MPI_Get_count(&status,datatype,&count);
Where it returns the number of items sent by the sender.

Point To Point Communication:

The most elementary form of message-passing communication involves two nodes, one passing a message to the other. Although there are several ways that this might happen in hardware, logically the communication is point-to-point: one node calls a **send** routine and the other calls a **receive**.

Blocking Send & Receive :

MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

buf-> address of the Send Buffer
count-> no. of items in the send buffer
Data Type describes the send items datatype.
dest-> rank of the destination process.
Tag is the message type identifier.
Comm is the communicator to be used.(doubt)

Note> the blocking send call does not return until it stored it's message away and to reuse the buffer again.

Blocking receive code syntax:

MPI_Recv(void* buf, int count, MPI_Datatype datatype,int source, int tag, MPI_Comm comm,MPI_Status *status);

All are the same except source and status: Source is the rank of the node sent by the message. status is an MPI-defined integer array of size MPI_STATUS_SIZE. It is used further by other routines.

Order: message passing maintains an order means if the sender sends two messages to the receiver and if the two messages match then the second message is not received by receiver until the first message is received completely.

Progress:
If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independent of other action in the system. The send operation will complete unless the receive is satisfied and completed by another message. The receive operation will complete unless the message sent is consumed by another matching receive that was posted at the same destination process.

Avoiding Deadlock:

This is a deadlock situation:-

```
MPI_Comm_rank(comm,&rank);
    if (rank == 0) {
      MPI_Recv(recvbuf,count,MPI_REAL,1,tag,comm,&status);
      MPI_Send(sendbuf,count,MPI_REAL,1,tag,comm);
    }
    elseif (rank == 1) {
      MPI_Recv(recvbuf,count,MPI_REAL,0,tag,comm,&status);
      MPI_Send(sendbuf,count,MPI_REAL,0,tag,comm);
    }
```

I understand it like the semaphore concept.
The receive operation of the first process must complete before its send, and can complete only if the matching send of the second process is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock.

### To Avoid Deadlock:

```
MPI_Comm_rank(comm,&rank);
    if (rank == 0) {
       MPI_Send(sendbuf,count,MPI_REAL,1,tag,comm,);
       MPI_Recv(recvbuf,count,MPI_REAL,1,tag,comm,&status);
    }
    elseif(rank == 1) {
       MPI_Recv(recvbuf,count,MPI_REAL,0,tag,comm,&status);
       MPI_Send(sendbuf,count,MPI_REAL,0,tag,comm);
    }
```

Or

```
MPI_Comm_rank(comm,&rank);
    if (rank == 0) {
       MPI_Recv(recvbuf,count,MPI_REAL,1,tag,comm,&status);
       MPI_Send(sendbuf,count,MPI_REAL,1,tag,comm);
    }
    elseif(rank == 1) {
       MPI_Send(sendbuf,count,MPI_REAL,0,tag,comm);
       MPI_Recv(recvbuf,count,MPI_REAL,0,tag,comm,&status);
    }
```

### MPI_Probe and Iprobe operation:

The MPI_Probe and MPI_Iprobe operations check for incoming messages without actually receiving them. After using these operations, the user can decide how to receive the messages, based on the information returned by the probe.
In particular, the user may allocate adequate memory for the receive buffer according to the length of the probed message. The syntax for both the blocking and nonblocking probes are as follows:

```
MPI_Probe(int source, int tag, MPI_Comm comm,MPI_Status *status);

MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);
```

### Gist of the Whole Document:

1.Blocking Communication is done using MPI_Send() and MPI_Recv(). This communication does not return until they finish, which means they block the messages.

2. In contrast Non-blocking communication uses MPI_Isend() and MPI_IRecv() which does not block. You have to use MPI_Wait() and MPI_Test()
3. Blocking send_receive calls are easy to use whereas Non blocking allows to overlap the computation hence it leads to better performance.