There are three possible parallel computers' memory architectures:

1. Shared memory
Uniform Memory Access (UMA)
Non-Uniform Memory Access (NUMA)

2. Distributed memory

3. Hybrid Distributed Shared memory

Distributed memory
In contrast to shared memory parallelism, in distributed memory parallelism, processes each keep their own private memories, separate from the memories of other processes.
In order for one process to access data from the memory of another process, the data must be communicated, commonly by a technique known as message passing, in which the data is packaged up and sent over a network.
In this architecture, the programmers have explicit control over data distribution and communication. Synchronization between tasks is programmer's responsibility.
One standard of message passing is the Message Passing Interface (MPI), which defines a set of functions that can be used inside of C, C++ or Fortran codes for passing messages.

The Message-Passing Interface (MPI)
supports a Distributed memory programming model
can be executed on distributed, shared or hybrid hardware platforms
is a message passing library standard, is a specification for the developers and users of message passing libraries
supports distributed parallelism, is used for developing message passing programs.
supports Explicit parallelism as programmers have explicit control over data distribution and communication and is responsible for identifying parallelism and implementing parallel applications. Synchronization between tasks is programmer's responsibility.
consists of a header file, a library of routines and a runtime environment.

Advantages of message-passing model

Portability- Programs need a little or no modification while porting to a different platform.
Provides the programmer with explicit control over the location of data in the memory.
Can be used on a wider range of problems than OpenMP..
Runs on distributed, shared or hybrid hardware platforms

Disadvantage of message-passing model

Extra effort required by the Programmer to convert program serial to parallel version.
Explicit parallelism makes debugging difficult, given the placement of memory and the ordering of communication requires additional details from the programmer.

MPI is actually just an Application Programming Interface (API).
An API specifies what a call to each routine should look like, and how each routine should behave.
An API does not specify how each routine should be implemented, and sometimes is intentionally vague about certain aspects of a routine's behavior.
Each platform has its own MPI implementation.

Minimal Set of MPI Routines

MPI_Init
starts up the MPI runtime environment at the beginning of a run.
MPI_Finalize
shuts down the MPI runtime environment at the end of a run.
MPI_Comm_size
gets the number of processes in a run, Np (typically called just after MPI_Init).
MPI_Comm_rank
gets the process ID that the current process uses, which is between 0 and Np-1 inclusive (typically called just after MPI_Init).
MPI_Send
sends a message from the current process to some other process (the destination).
MPI_Recv
receives a message on the current process from some other process (the source).
MPI_Bcast
broadcasts a message from one process to all of the others.
MPI_Reduce
performs a reduction (for example, sum, maximum) of a variable on all processes, sending the result to a single process.

MPI Program Structure (C)

```
#include <stdio.h>
#include "mpi.h"
 [other includes]
int main (int argc, char* argv[])
{ /* main */
  int my_rank, num_procs, mpi_error_code;
   [other declarations]
  mpi_error_code =
    MPI_Init(&argc, &argv);        /* Start up MPI   */
  mpi_error_code =
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error_code =
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
   [actual work goes here]
  mpi_error_code = MPI_Finalize(); /* Shut down MPI */
} /* main */
```

MPI is Multiple Program, Multiple data (MPMD) model:
each processor runs independently of the others with independent programs and data, and a different instruction sequences on different data sets are executed simultaneously on a set of processors.
To make job of programmer easy and achieve scalability most of the message passing programs are written using single program multiple data (SPMD) approach.
Processes can use:
point-to-point communication operations to send a message from one named process to another.
collective communication operations to collectively perform commonly used global operations such as summation and broadcast.

An MPI communicator is a collection of processes that can send messages to each other.
MPI_COMM_WORLD is the default communicator; it contains all of the processes. It's probably the only one you'll need.
Some libraries create special library-only communicators, which can simplify keeping track of message tags.

What happens if one process has data that everyone else needs to know?
For example, what if the server process needs to send an input value to the others?
MPI_Bcast(length, 1, MPI_INTEGER,
  source, MPI_COMM_WORLD);

Note that MPI_Bcast doesn't use a tag, and that the call is the same for both the sender and all of the receivers.
All processes have to call MPI_Bcast at the same time; everyone waits until everyone is done.

A reduction converts an array to a scalar: for example sum, product, minimum value, maximum value, Boolean AND, Boolean OR, etc.

Reductions are so common, and so important, that MPI has two routines to handle them:
MPI_Reduce: sends result to a single specified process
MPI_Allreduce: sends result to all processes (and therefore takes longer)

MPI allows a process to start a send, then go on and do work while the message is in transit.
This is called non-blocking or immediate communication.
Here, "immediate" refers to the fact that the call to the MPI routine returns immediately rather than waiting for the communication to complete.

```
mpi_error_code =
   MPI_Isend(array, size, MPI_FLOAT,
      destination, tag, communicator, request);
```

Likewise:

```
mpi_error_code =
   MPI_Irecv(array, size, MPI_FLOAT,
      source, tag, communicator, request);
```

This call starts the send/receive, but the send/receive won't be complete until:

```
MPI_Wait(request, status);
```

In between the call to MPI_Isend/Irecv and the call to MPI_Wait, both processes can do work!
If that work takes at least as much time as the communication, then the cost of the communication is effectively zero, since the communication won't affect how much work gets done.
This is called communication hiding.