Nicola Spallanzani
n.spallanzani@cineca.it
High Performance Computing department

Parallel programming with MPI

Collective Communications

Collective communications

Collective communications is a method of communication which involves all processes in a communicator:

- All processes (in a communicator) call the collective function
- Collective communications will not interfere with point-to-point
- All collective communications are blocking (in MPI 2.0)
- No tags are required
- Receive buffers must match in size (number of bytes)

It’s a safe communication mode

Collective communications

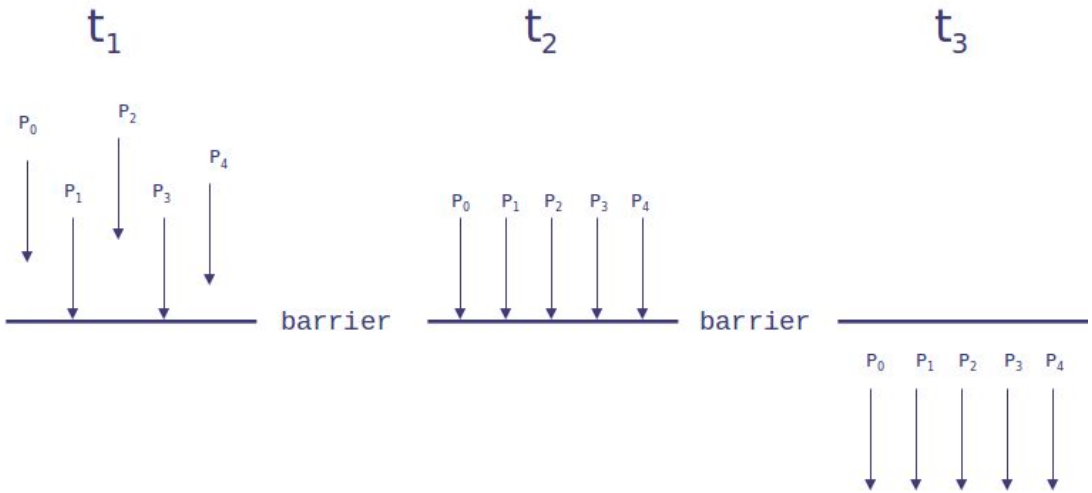
Communications involving a group of processes. They are called by all the ranks involved in a communicator (or a group) and are of three types:

- Synchronization (e.g. Barrier)
- Data Movement (e.g. Broadcast or Gather/scatter)
- Global Computation (e.g. reductions)

MPI Barrier

It stops all processes within a communicator until they are synchronized

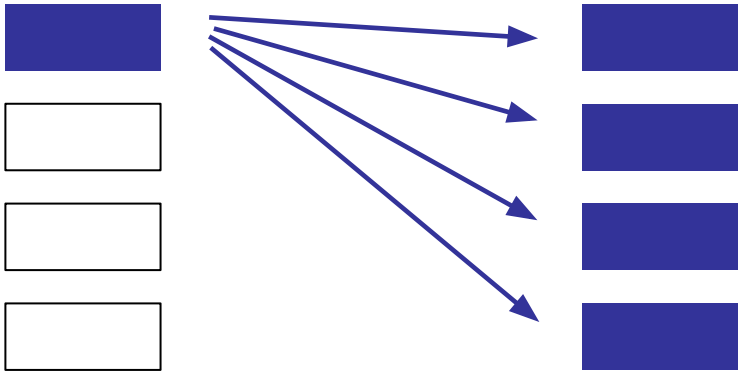
```
int MPI_Barrier(MPI_Comm comm);
```



MPI Broadcast

```
int MPI_Bcast (void *buf, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm);
```

Note that all processes must specify the same root and same comm.



Example

```
PROGRAM broad_cast
     USE MPI
     INTEGER :: ierr, myid, nproc, root
     INTEGER :: status(MPI_STATUS_SIZE)
     REAL :: A(2)
     CALL MPI_INIT(ierr)
     CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
     root = 0
     IF( myid .EQ. root ) THEN
          a(1) = 2.0
          a(2) = 4.0
     END IF
     CALL MPI_BCAST(a, 2, MPI_REAL, root, MPI_COMM_WORLD, ierr)
     WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
     CALL MPI_FINALIZE(ierr)
END PROGRAM broad_cast
```

MPI Gather

Each process, root included, sends the content of its send buffer to the root process. The root process receives the messages and stores them in the rank order.

```
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
               void *recvbuf, int recvcnt, MPI_Datatype recvtype,
               int root, MPI_Comm comm);
```

MPI Scatter

The root sends a message. The message is split into n equal segments, the i-th segment is sent to the i-th process in the group and each process receives this message.

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
```

MPI Allgather

There are possible combinations of collective functions.

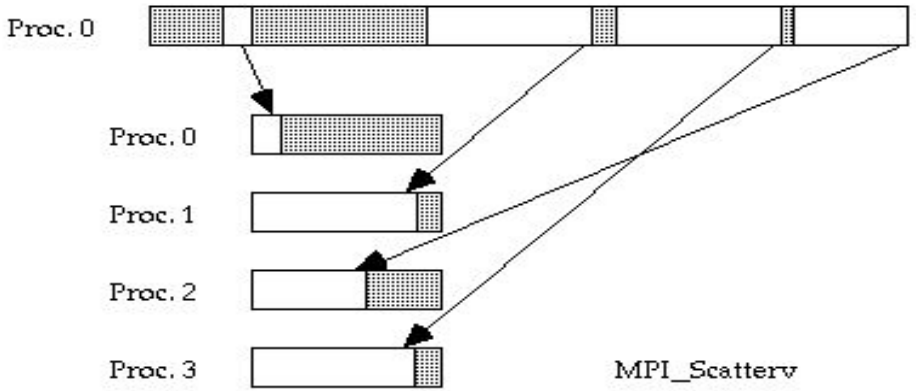
For example, MPI Allgather is a combination of a gather + a broadcast:

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm);
```

MPI Scatterv, MPI Gatherv

For many collective functions there are extended functionalities.

For example it’s possible to define the length of arrays to be scattered or gathered with MPI_Scatterv and MPI_Gatherv.



MPI Scaterv

C:

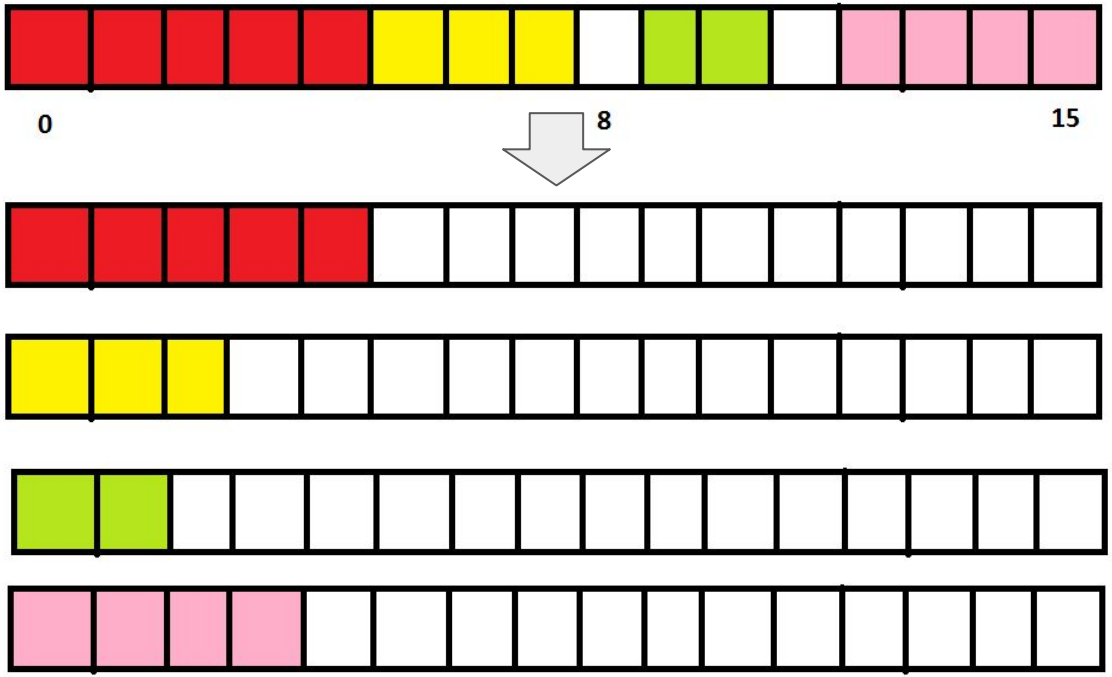
```
int MPI_Scatterv(const void *sendbuf, const int sendcounts[],
    const int displs[], MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

Fortran:

```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE,
             RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR)
INTEGER    SENDCOUNTS(*), DISPLS(*), SENDTYPE
INTEGER    RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

MPI Scaterv

sendcounts=(5,3,2,4) displs=(0,5,9,12)



MPI Gatherv

C:

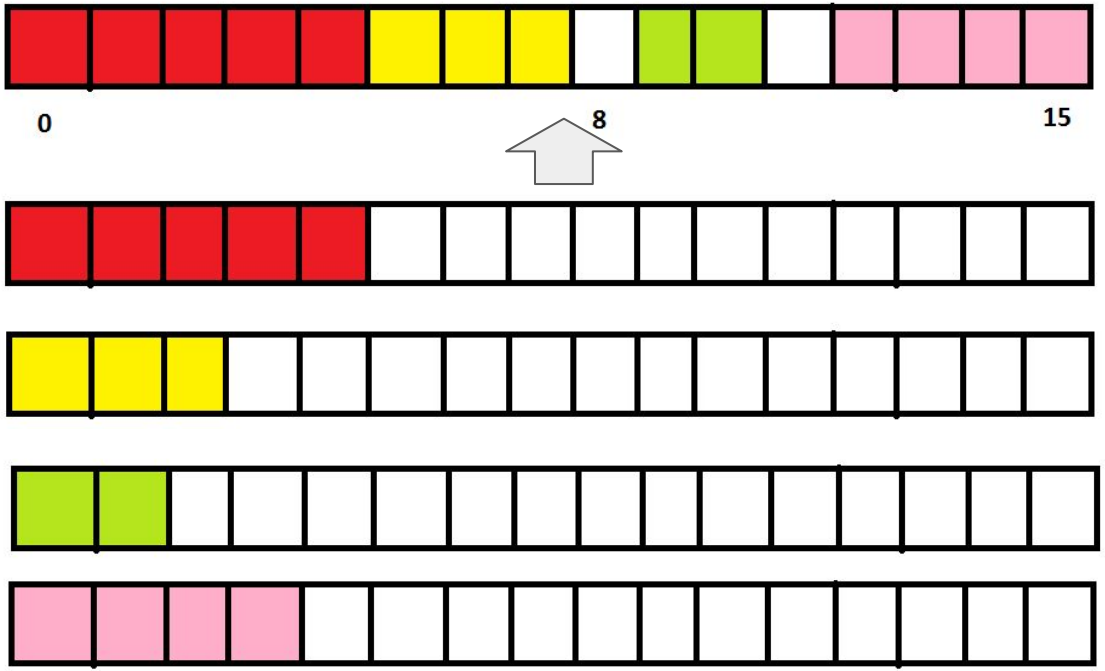
```
int MPI_Gatherv(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
    const int displs[], MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

Fortran:

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE,
            RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE,
            ROOT, COMM, IERROR)
INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*)
INTEGER    RECVTYPE, ROOT, COMM, IERROR
```

MPI Gatherv

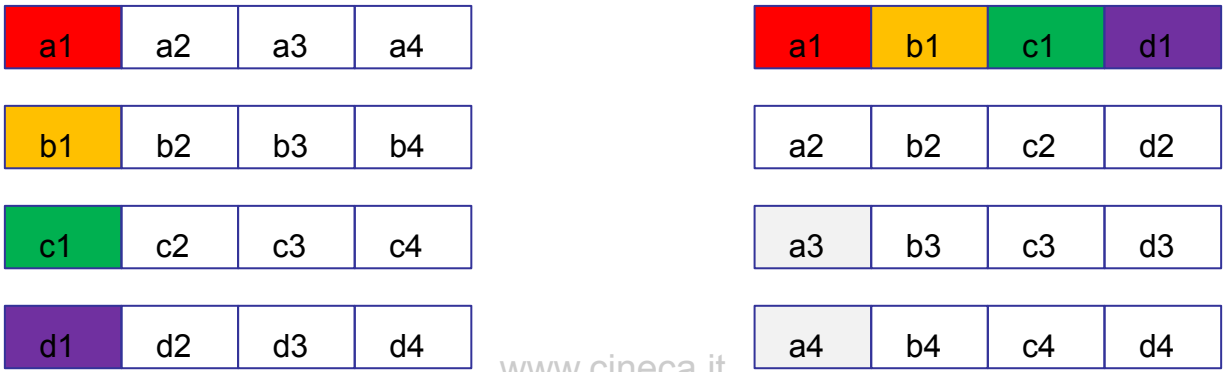
recvcounts=(5,3,2,4) displs=(0,5,9,12)



MPI All to All

This function makes a redistribution of the content of each process in a way that each process know the buffer of all others. It is a way to implement the matrix data transposition.

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```



Reduction

Reduction operations permits us to

- Collect data from each process
- Reduce the data to a single value
- Store the result on the root process (MPI_Reduce) or
- Store the result on all processes (MPI_Allreduce)

C:

```
int MPI_Reduce(const void *sendbuf,
    void *recvbuf, int count,
    MPI_Datatype datatype,
    MPI_Op op, int root,
    MPI_Comm comm);
```

Predefined reduction operations

MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Example

```
PROGRAM reduce
     USE MPI
     INTEGER :: ierr, myid, nproc, root
     REAL :: A(2), res(2)
     CALL MPI_INIT(ierr)
     CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
     root = 0
     A(1) = 2.0 * myid
     A(2) = 4.0 * myid
     CALL MPI_REDUCE(A, res, 2, MPI_REAL, MPI_SUM, root, MPI_COMM_WORLD, ierr)
     IF( myid .EQ. 0 ) THEN
          WRITE(6,*) myid, ': res(1)=', res(1), 'res(2)=', res(2)
     END IF
     CALL MPI_FINALIZE(ierr)
END
```

Performance issues

- Much hidden communication takes place with collective communication.
- Hardware vendors work hard to provide optimized collective calls but performances will vary according to implementation.
- Because of forced synchronization, collective communications may not always be the best solution.

Some studies show that around 80% transfer time is in collectives.