epcc

Collective Communications

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronisation.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

- Collective action over a communicator.
- All processes must communicate.
- Synchronisation may or may not occur.
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size.

- C:

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

- Fortran:

```
MPI_BARRIER (COMM, IERROR)
INTEGER COMM, IERROR
```

- C:

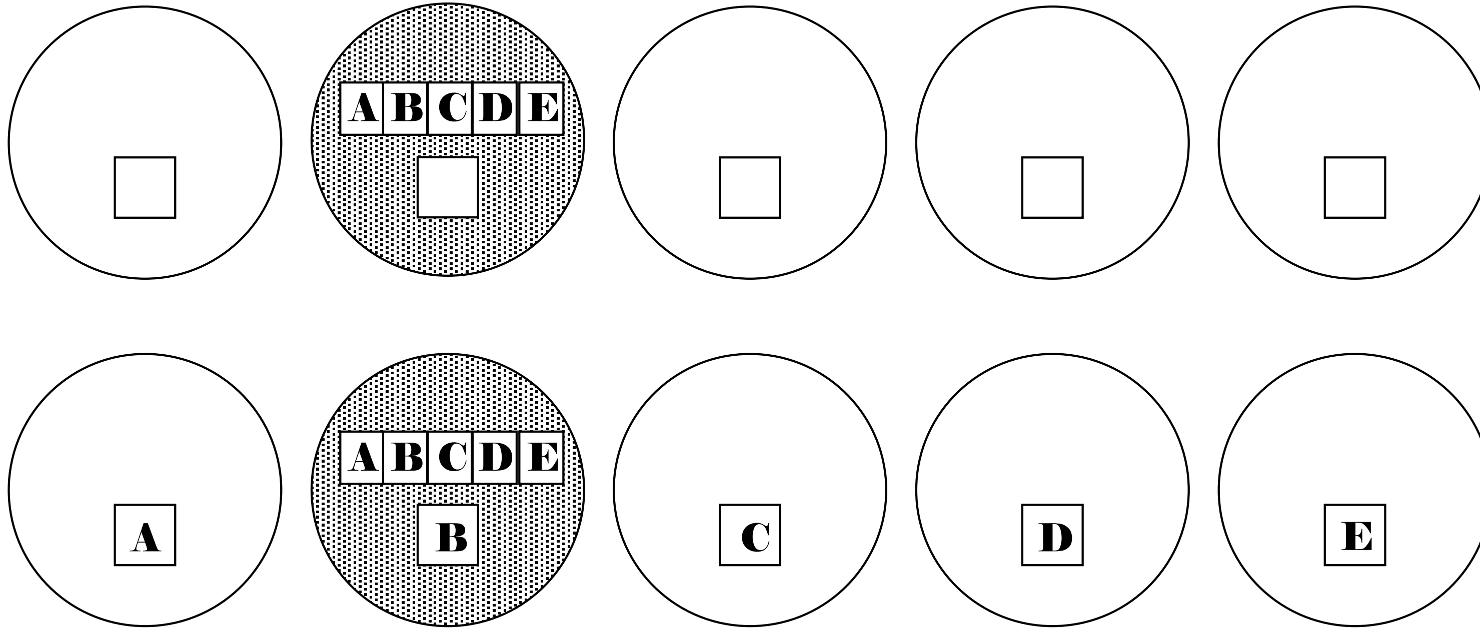
```
int MPI_Bcast (void *buffer, int count,
               MPI_Datatype datatype, int root,
               MPI_Comm comm)
```

- Fortran:

```
MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT,
           COMM, IERROR)

<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

Scatter



- C:

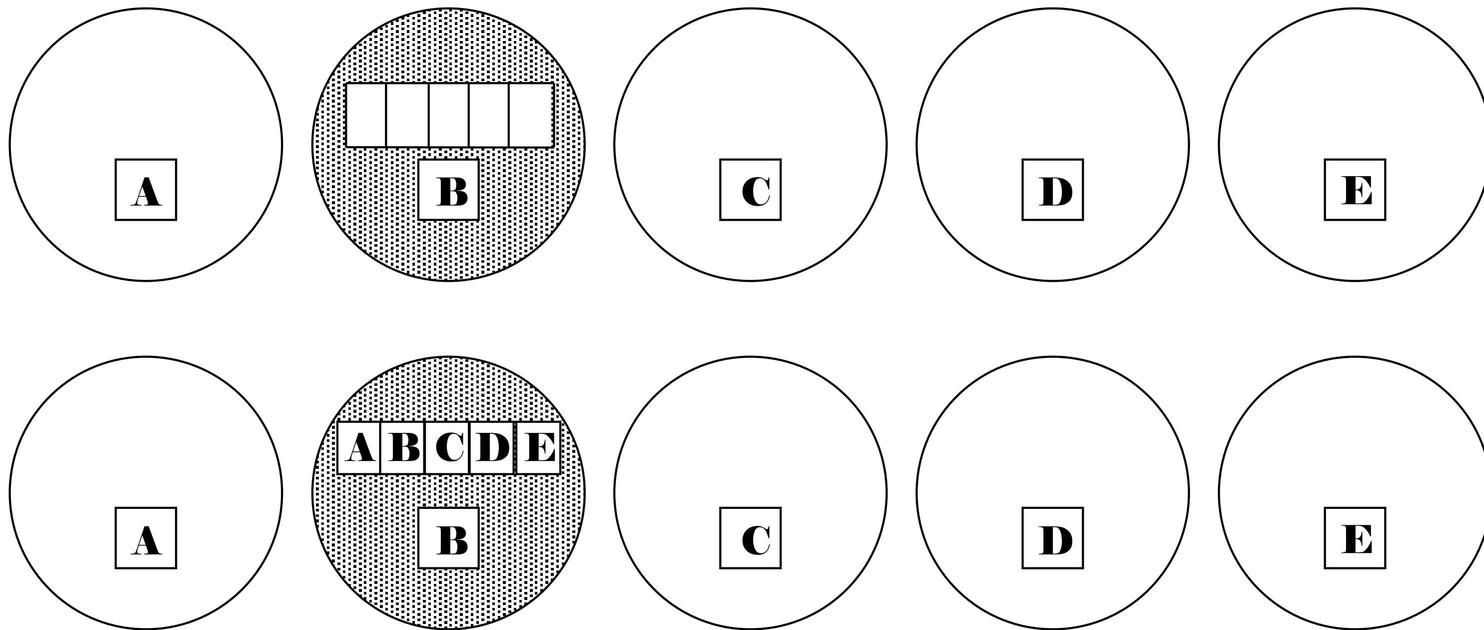
```
int MPI_Scatter(void *sendbuf,
     int sendcount, MPI_Datatype sendtype,
     void *recvbuf, int recvcount,
     MPI_Datatype recvtype, int root,
     MPI_Comm comm)
```

- Fortran:

```
     MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE,
                 RECVBUF, RECVCOUNT, RECVTYPE,
                 ROOT, COMM, IERROR)

     <type> SENDBUF, RECVBUF
     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
     INTEGER RECVTYPE, ROOT, COMM, IERROR
```

Gather



Gather

- C:

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

- Fortran:

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE,
          RECVBUF, RECVCOUNT, RECVTYPE,
          ROOT, COMM, IERROR)

<type> SENDBUF, RECVBUF
INTEGER   SENDCOUNT, SENDTYPE, RECVCOUNT
 INTEGER RECVTYPE, ROOT, COMM, IERROR
```

- Used to compute a result involving data distributed over a group of processes.
- Examples:
  - global sum or product
  - global maximum or minimum
  - global user-defined operation

Predefined Reduction Operations

MPI Name	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

- C:

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

- Fortran:

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT,
           DATATYPE, OP, ROOT, COMM, IERROR)

    <type> SENDBUF, RECVBUF
    INTEGER     SENDCOUNT, SENDTYPE, RECVCOUNT
    INTEGER RECVTYPE, ROOT, COMM, IERROR
```

MPI_REDUCE

Rank

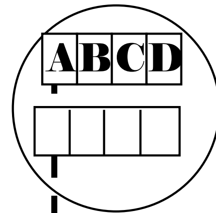
0

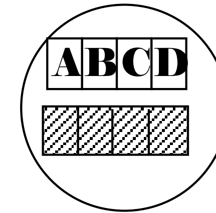
1

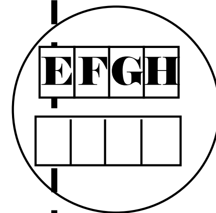
2

3

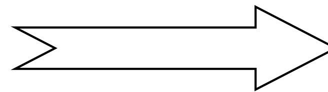
Root

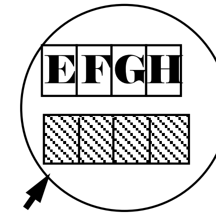


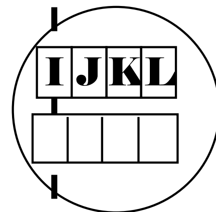


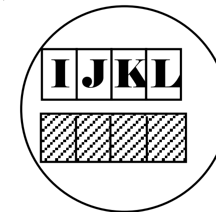


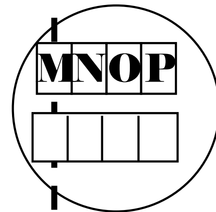
MPI_REDUCE

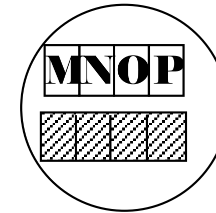




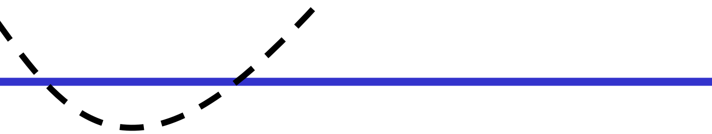








AoEoIoM



Integer global sum

- C:

```
MPI_Reduce(&x, &result, 1, MPI_INT,
           MPI_SUM,0, MPI_COMM_WORLD)
```

- Fortran:

```
CALL MPI_REDUCE(x, result, 1, MPI_INTEGER,
                MPI_SUM, 0,
                MPI_COMM_WORLD, IERROR)
```

- Sum of all the *x* values is placed in *result*.
- The result is only placed there on processor 0.

- Reducing using an arbitrary operator, ∘
- C - function of type MPI_User_function:

```
void my_op (void *invec,
           void *inoutvec,int *len,
           MPI_Datatype *datatype)
```

- Fortran - external subprogram of type

```
SUBROUTINE MY_OP(INVEC(*),INOUTVEC(*),
                LEN, DATATYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, DATATYPE
```

- Operator function for ○ must act as:

```
for (i = 1 to len)
      inoutvec(i) = inoutvec(i) o invec(i)
```

- Operator ○ need not commute but must be associative.

- Operator handles have type `MPI_Op` or `INTEGER`
- C:

```
int MPI_Op_create(MPI_User_function *my_op,
                  int commute, MPI_Op *op)
```

- Fortran:

```
MPI_OP_CREATE (MY_OP, COMMUTE, OP, IERROR)

EXTERNAL MY_OP
LOGICAL COMMUTE
INTEGER OP, IERROR
```

- `MPI_Allreduce` no root process
- `MPI_Reduce_scatter` result is scattered
- `MPI_Scan` “parallel prefix”

MPI_ALLREDUCE

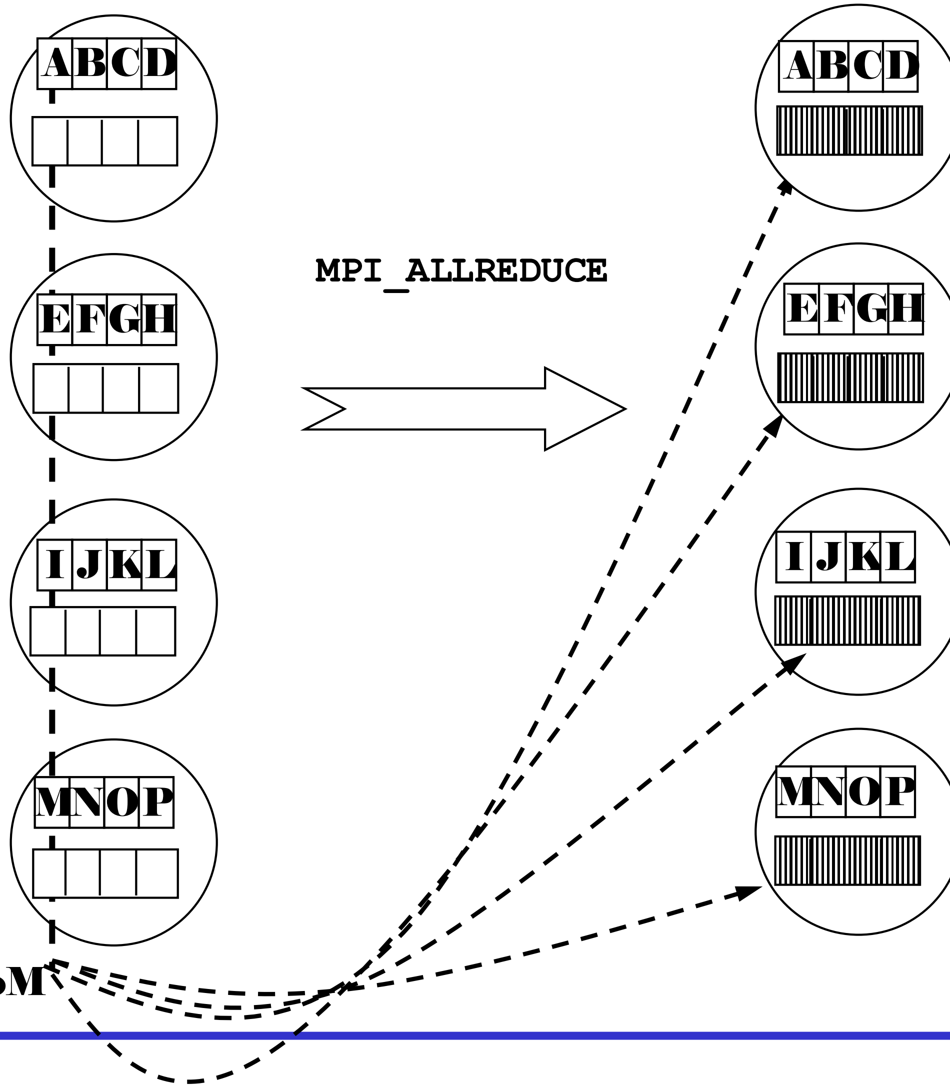
Rank

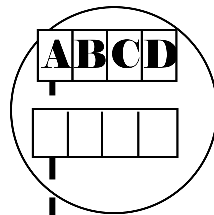
0

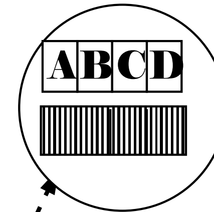
1

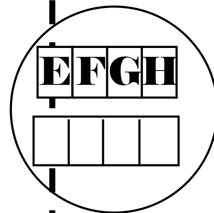
2

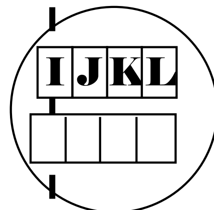
3

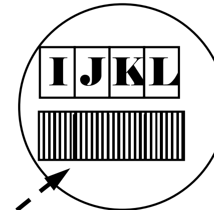


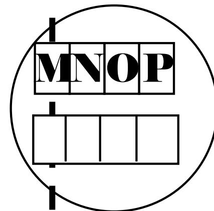


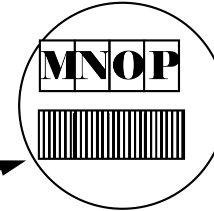


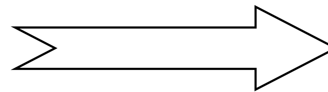












Integer global sum

- C:

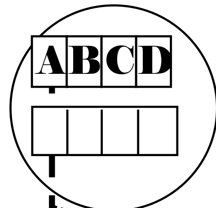
```
int MPI_Allreduce(void* sendbuf,
          void* recvbuf, int count,
          MPI_Datatype datatype,
          MPI_Op op, MPI_Comm comm)
```

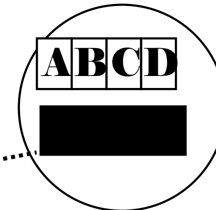
- Fortran:

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT,
            DATATYPE, OP, COMM, IERROR)
```

Rank

0

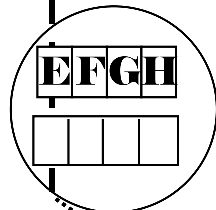


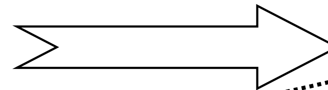


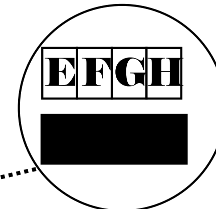
A

MPI_SCAN

1

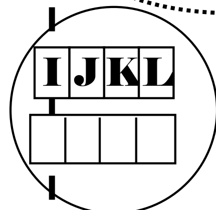


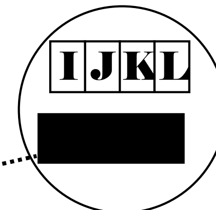




AoE

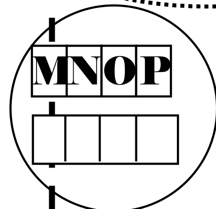
2

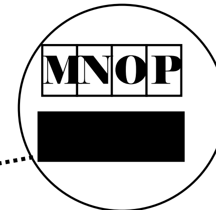




AoEoI

3





AoEoIoM

Integer partial sum

- C:

```
int MPI_Scan(void* sendbuf, void* recvbuf,
         int count, MPI_Datatype datatype,
         MPI_Op op, MPI_Comm comm)
```

- Fortran:

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT,
      DATATYPE, OP, COMM, IERROR)
```

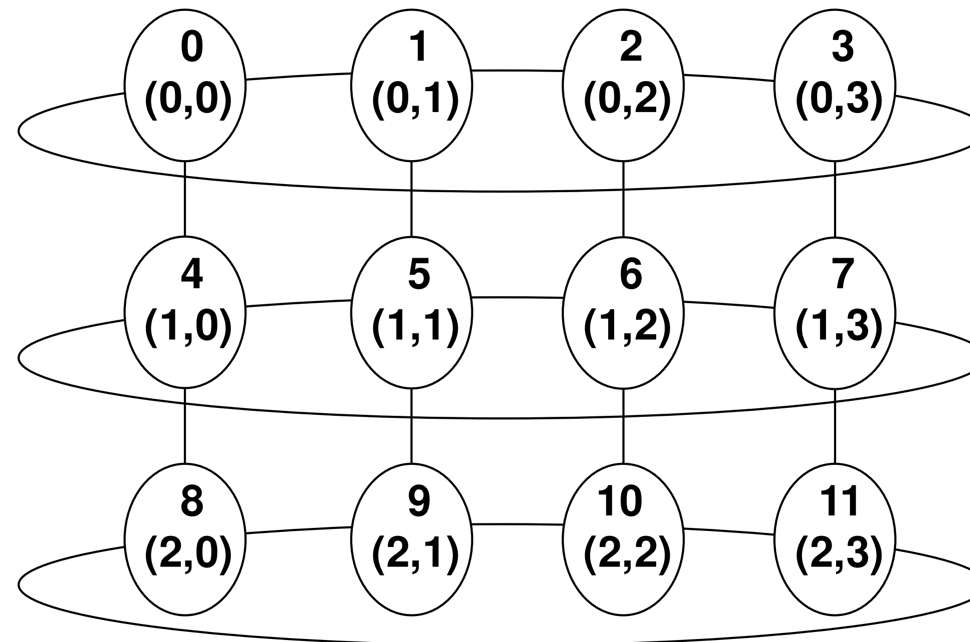
epcc

Virtual Topologies

- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimise communications.

- Creating a topology produces a new communicator.
- MPI provides ``mapping functions''.
- Mapping functions compute processor ranks, based on the topology naming scheme.

A 2-dimensional Cylinder



- Cartesian topologies
  - each process is “connected” to its neighbours in a virtual grid.
    - boundaries can be cyclic, or not.
    - optionally re-order ranks to allow MPI implementation to optimise for underlying network interconnectivity.
  - processes are identified by cartesian coordinates.
- Graph topologies
  - general graphs
  - not covered here

Creating a Cartesian Virtual Topology

- C:

```
int MPI_Cart_create(MPI_Comm comm_old,
          int ndims, int *dims, int *periods,
          int reorder, MPI_Comm *comm_cart)
```

- Fortran:

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS,
          PERIODS, REORDER, COMM_CART, IERROR)

INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL PERIODS(*), REORDER
```

- C:

```
int MPI_Dims_create(int nnodes, int ndims,
                    int *dims)
```

- Fortran:

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)

INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

MPI_Dims_create

- Call tries to set dimensions as close to each other as possible

dims before the call	function call	dims on return
(0, 0)	MPI_DIMS_CREATE(6, 2, dims)	(3, 2)
(0, 0)	MPI_DIMS_CREATE(7, 2, dims)	(7, 1)
(0, 3, 0)	MPI_DIMS_CREATE(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

- Non zero values in dims sets the number of processors required in that direction.
  - WARNING:- make sure dims is set to 0 before the call!

Mapping process grid coordinates to ranks

- C:

```
int MPI_Cart_rank(MPI_Comm comm,
                  int *coords, int *rank)
```

- Fortran:

```
MPI_CART_RANK (COMM, COORDS, RANK, IERROR)

INTEGER COMM, COORDS(*), RANK, IERROR
```

Mapping ranks to process grid coordinates

- C:

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
                    int maxdims, int *coords)
```

- Fortran:

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS,COORDS,
                IERROR)

INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

Computing ranks of my neighbouring processes
Following conventions of MPI_SendRecv

- C:

```
int MPI_Cart_shift(MPI_Comm comm,
                   int direction, int disp,
                   int *rank_source, int *rank_dest)
```

- Fortran:

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP,
               RANK_SOURCE, RANK_DEST, IERROR)

INTEGER COMM, DIRECTION, DISP,
        RANK_SOURCE, RANK_DEST, IERROR
```

Non-existent ranks

- What if you ask for the rank of a non-existent process?
  - or look off the edge of a non-periodic grid?
- MPI returns a NULL processor
  - rank is MPI_PROC_NULL
- MPI_PROC_NULL is a black hole
  - sends and receives complete immediately
  - send buffer disappears, receive buffer isn’t touched
  - like UNIX /dev/null

- Cut a grid up into “slices”.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.
- `MPI_Cart_sub` and `MPI_CART_SUB` generate new communicators for the slices.
  – Use array to specify which dimensions should be retained in the new communicator.

- C:

```
int MPI_Cart_sub (MPI_Comm comm,
        int *remain_dims,
        MPI_Comm *newcomm)
```

- Fortran:

```
MPI_CART_SUB (COMM, REMAIN_DIMS,
              NEWCOMM,IERROR)

INTEGER COMM, NEWCOMM, IERROR
LOGICAL REMAIN_DIMS(*)
```

```c
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT  2
#define RIGHT 3

main(int argc, char *argv[])  {
int numtasks, rank, source, dest, outbuf, i, tag=1,
   inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
   nbrs[4], dims[2]={4,4},
   periods[2]={0,0}, reorder=0, coords[2];

MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;   // required variable

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
   // create cartesian virtual topology, get rank, coordinates, neighbor ranks
   MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
   MPI_Comm_rank(cartcomm, &rank);
   MPI_Cart_coords(cartcomm, rank, 2, coords);
   MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
   MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

   printf("rank= %d coords= %d %d  neighbors(u,d,l,r)= %d %d %d %d\n",
          rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
          nbrs[RIGHT]);

   outbuf = rank;

   // exchange data (rank) with 4 neighbors
   for (i=0; i<4; i++) {
      dest = nbrs[i];
      source = nbrs[i];
      MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
                MPI_COMM_WORLD, &reqs[i]);
      MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
                MPI_COMM_WORLD, &reqs[i+4]);
      }

   MPI_Waitall(8, reqs, stats);

   printf("rank= %d                  inbuf(u,d,l,r)= %d %d %d %d\n",
          rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);  }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Sample program output: (partial)

```
rank=   0 coords=  0 0 neighbors(u,d,l,r)=  -1  4 -1  1
rank=   0                    inbuf(u,d,l,r)=  -1  4 -1  1
rank=   8 coords=  2 0 neighbors(u,d,l,r)=   4 12 -1  9
rank=   8                    inbuf(u,d,l,r)=   4 12 -1  9
rank=   1 coords=  0 1 neighbors(u,d,l,r)=  -1  5  0  2
rank=   1                    inbuf(u,d,l,r)=  -1  5  0  2
rank=  13 coords=  3 1 neighbors(u,d,l,r)=   9 -1 12 14
rank=  13                    inbuf(u,d,l,r)=   9 -1 12 14
...
...
rank=   3 coords=  0 3 neighbors(u,d,l,r)=  -1  7  2 -1
rank=   3                    inbuf(u,d,l,r)=  -1  7  2 -1
rank=  11 coords=  2 3 neighbors(u,d,l,r)=   7 15 10 -1
rank=  11                    inbuf(u,d,l,r)=   7 15 10 -1
rank=  10 coords=  2 2 neighbors(u,d,l,r)=   6 14  9 11
rank=  10                    inbuf(u,d,l,r)=   6 14  9 11
rank=   9 coords=  2 1 neighbors(u,d,l,r)=   5 13  8 10
rank=   9                    inbuf(u,d,l,r)=   5 13  8 10
```

Calendar till Minor 2

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Interconnection, MPI	10	11 **assign2 released**	12	13	14	15	16
Synchronization	17	18 **assign1, minor1**	19	20	21	22	23
Memory consistency	24	25	26	27	28	29	1
	2	3 **assign2 due**	4	5	6	7	8
Problem solving	9	10	11	12	13	14	15
	16	17	18	19	20	21	22