Introduction to Parallel Computing
MPI – Message Passing Interface

Vistas in Advanced Computing / Summer 2017

UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Terminology (I)

- an `MPI_Group` is the object describing the list of processes forming a logical entity
  - a group has a size `MPI_Group_size`
  - every process in the group has a unique rank between 0 and (size of group -1) `MPI_Group_rank`
  - a group is a local object, and cannot be used for any communication

Terminology (II)

- An `MPI_Comm`(unicator) is an object containing
  - one or two groups of processes (*intra* or *inter*-communicators)
  - Context
  - topology information
  - attributes
- A communicator has an error handler attached to it
- A communicator can have a name
- these slides focus on *intra*-communicators i.e. the list of participating processes can be described by a single group

Predefined communicators

- `MPI_COMM_WORLD`
  - contains all processes started with `mpirun/mpiexec`
  - exist upon exiting `MPI_Init`
  - can not be modified, freed etc.

- `MPI_COMM_SELF`
  - contains just the local process itself, size is always 1
  - exist upon exiting `MPI_Init`
  - can not be modified, freed etc.

Creating new communicators

- All communicators in MPI-1 are derived from `MPI_COMM_WORLD` or `MPI_COMM_SELF`
- Creating and freeing a communicator is a **collective** operation ➔ all processes of the original communicator have to call the function with the same arguments
- Methods to create new communicators
  - splitting the original communicator into n-parts
  - creating subgroups of the original communicator
  - re-ordering of processes based on topology information

  - spawn new processes
  - connect two applications and merge their communicators

Splitting a communicator

```
MPI_Comm_split ( MPI_Comm comm, int color,
                 int key, MPI_comm *newcomm);
```

- Partition `comm` into sub-communicators
  - all processes having the same `color` will be in the same subcommunicator
  - order processes with the same `color` according to the `key` value
  - if the `key` value is identical on all processes with the same `color`, the same order for the processes will be used as in `comm`

Example for MPI_Comm_split (I)

```
MPI_Comm newcomm;
int color, rank;

MPI_Comm_rank (MPI_COMM_WORLD, &rank);
color = rank%2;

MPI_Comm_split (MPI_COMM_WORLD, color, rank, &newcomm);
MPI_Comm_size (newcomm, &size);
MPI_Comm_rank (newcomm, &rank);
```

- odd/even splitting of processes
- a process
  - can just be part of **one** of the generated communicators
  - can not “see” the other communicators
  - can not “see” how many communicators have been created

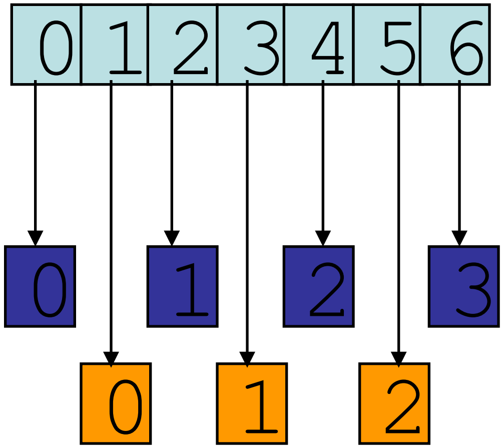
Example for MPI_Comm_split (II)

- rank and size of the new communicator

`MPI_COMM_WORLD`

`newcomm, color=0, size = 4`

`newcomm, color=1, size = 3`

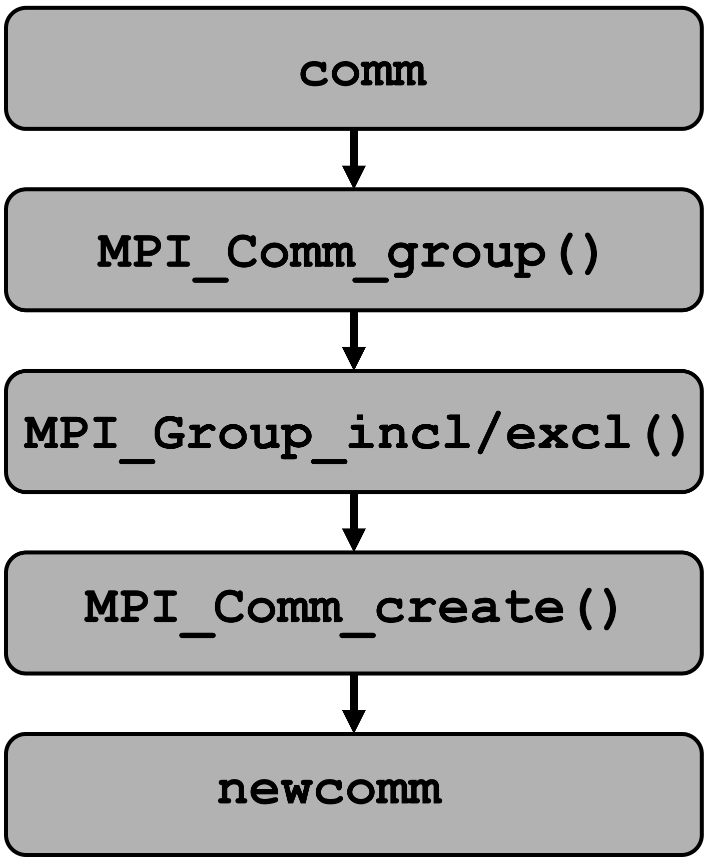


Invalid color in MPI_Comm_split

- If a process shall not be part of any of the resulting communicators
  - set `color` to `MPI_UNDEFINED`
  - `newcomm` will be `MPI_COMM_NULL`

- `MPI_COMM_NULL` is an invalid communicator
  - any function taking a communicator as an argument will return an error (or abort) if you pass `MPI_COMM_NULL`
  - i.e. even `MPI_Comm_size` and `MPI_Comm_rank`, or `MPI_Comm_free`

Modifying the group of processes



original communicator

Extract the group of processes from the original communicator

Modify the group

Create new communicator based on the modified group

new communicator

Extracting the group of processes

```
MPI_Comm_group (MPI_Comm comm, MPI_Group *group);
```

with

- `comm`: original communicator
- `group`: the group object describing the list of participating processes in `comm`

Modifying groups (I)

```
MPI_Group_incl (MPI_Group group, int cnt, int ranks[],
                MPI_Group *newgroup);
MPI_Group_excl (MPI_Group group, int cnt, int ranks[],
                MPI_Group *newgroup);
```

with

- `group`: the original group object containing the list of participating processes
- `ranks[]` : array of integers containing the ranks of the processes in group, which shall be
  - included in the new group for `MPI_Group_incl`
  - excluded from the original `group` for `MPI_Group_excl`
- `newgroup`: resulting group

Modifying groups (II)

- for more group-constructors, see also
  - `MPI_Group_range_incl`
  - `MPI_Group_range_excl`
  - `MPI_Group_difference`
  - `MPI_Group_intersection`
  - `MPI_Group_union`

Creating a new communicator based on a group

```
MPI_Comm_create ( MPI_Comm comm, MPI_Group newgroup,
                  MPI_Comm *newcomm);
```

with

- `comm`: original communicator
- `group`: the group object describing the list of processes for the new communicator
- `newcomm`: resulting communicator

Note:

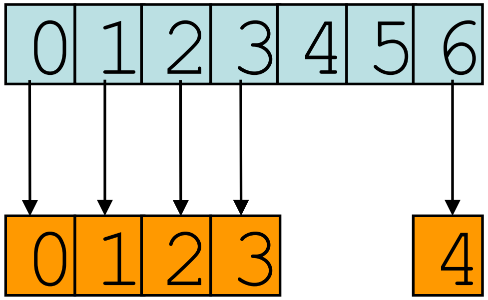
- `newcomm` is always a subset of `comm`
- you can generate **one** communicator at a time (in contrary to `MPI_Comm_split`)
  - list of arguments has to be identical on all processes of comm
- `newcomm` will be `MPI_COMM_NULL` for processes which have been excluded/not included in `newgroup`

Example for MPI_Comm_create

- generate a communicator, which contains only the first four processes and the last process of the original communicator

`MPI_COMM_WORLD`

`newcomm, size = 5`



1st Option: using MPI_Group_incl

```
MPI_Comm newcomm;
MPI_Group group, newgroup;
int color, size, ranks[5], cnt;

MPI_Comm_size (MPI_COMM_WORLD, &size);

cnt = 5;
ranks[0] = 0; ranks[1] = 1; ranks[2] = 2; ranks[3] = 3;
ranks[4] = size-1

MPI_Comm_group (MPI_COMM_WORLD, &group);
MPI_Group_incl (group, cnt, ranks, &newgroup)
MPI_Comm_create (MPI_COMM_WORLD, newgroup, &newcomm);
if ( newcomm != MPI_COMM_NULL ) {
       MPI_Comm_rank (newcomm, &nrank);
       MPI_Comm_free (&newcomm);
       MPI_Group_free (&newgroup);
}
MPI_Group_free (&group);
```

2nd Option: using MPI_Group_excl

```
MPI_Comm newcomm;
MPI_Group group, newgroup;
int color, size, ranks[…], cnt;
/* NOTE: Assuming that size >5, ranks is large enough
etc.  */
MPI_Comm_size (MPI_COMM_WORLD, &size);

cnt = 0;
for ( i=4; i<(size-1); i++) {
   ranks[cnt++] = i;
}

MPI_Comm_group (MPI_COMM_WORLD, &group);
MPI_Group_excl (group, cnt-1, ranks, &newgroup)
MPI_Comm_create (comm, newgroup, &newcomm);
if ( newcomm != MPI_COMM_NULL ) {
       MPI_Comm_rank (newcomm, &nrank);
       MPI_Comm_free (&newcomm);
       MPI_Group_free (&newgroup);
}
MPI_Group_free (&group);
```

Freeing groups and communicators

```
MPI_Comm_free ( MPI_Comm *comm);
MPI_Group_free ( MPI_Group *group);
```

- return `MPI_COMM_NULL` respectively `MPI_GROUP_NULL`
- `MPI_Comm_free` is a collective function,
- `MPI_Group_free` is a local function

Topology information in communicators

- Some application scenarios require not only to know who is part of a communicator but also how they are organized
  - Called topology information
  - 1-D, 2-D,3-D,… cartesian topology
  - What are the extent of each dimensions
  - Who are my left/right, upper/lower neighbors etc…
- Yes, its easy to do that yourself in the application
  - Position x-direction: $coord_x = rank \% n_x$
  - Position in y-direction: $coord_y = floor (rank / n_x)$

MPI_Cart_create

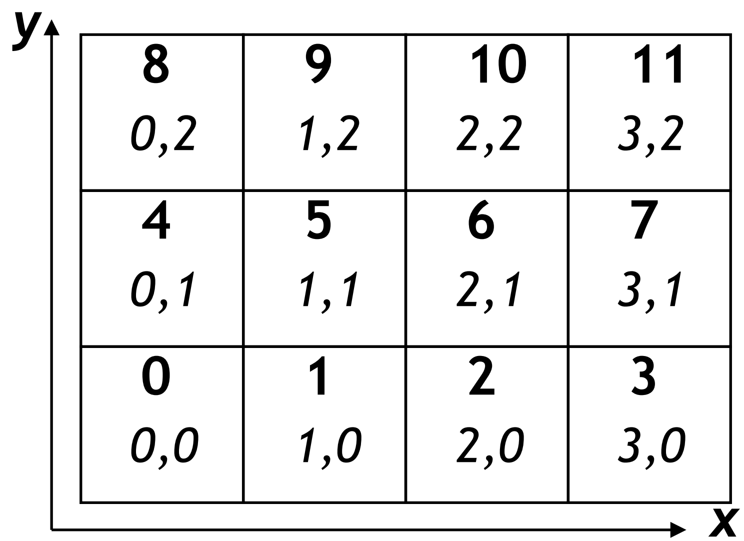
```
MPI_Cart_create ( MPI_Comm comm, int ndims,
                  int *dims, int *periods,
                  int reorder, MPI_Comm *newcomm);
```

- Create a new communicator having a cartesian topology with
  - `ndims` dimensions
  - Each dimension having `dims[i]` processes, `i=0,…,ndims-1`
  - `periods[i]` indicates whether the boundaries for the i-th dimension are wrapped around or not
  - `reorder`: flag allowing the MPI library to rearrange processes
- Note: if $\prod_{i=0}^{ndims-1} dims[i] <$ size of `comm`, some processes will not be part of newcomm

UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Example for using MPI_Cart_create

- Consider an application using 12 processes and arranging the processes in a 2-D cartesian topology




```
int ndims=2;
int dims[2]= {4,3};
int periods[2] = {0,0}; // no periodic boundaries
int reorder=0;          // no reordering of processes
MPI_Comm newcomm;

MPI_Cart_create ( MPI_COMM_WORLD, ndims, dims, periods,
                  reorder, &newcomm);
```

Who are my neighbors?

- easy to determine by hand for low dimensions, e.g.

np_x : no of procs in *x* direction

np_y : no of procs in *y* direction

$$n_{left} = rank - 1$$

$$n_{right} = rank + 1$$

$$n_{up} = rank + np_x$$

$$n_{down} = rank - np_x$$

- more complex for higher dimensional topologies
- special care needed at the boundaries

Who are my neighbors?

```
MPI_Cart_shift ( MPI_Comm comm, int direction,
                 int distance, int *leftn,
                 int *rightn);
```

- with
  - `direction`: dimension for which you would like to determine the ranks of the neighboring processes
  - `distance`: distance between the current process and the neighbors that you are interested in
  - `leftn`: rank of the left neighbor in `comm`
  - `rightn`: rank of the right neighbor in `comm`
- if a process does not have a left/right neighbor (e.g. at the boundary), `leftn` and/or `rightn` will contain `MPI_PROC_NULL`

UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Example for using MPI_Cart_shift

- continuing the example from MPI_ Cart_create

```
int ndims=2;
int dims[2]= {4,3};
int periods[2] = {0,0}; // no periodic boundaries
int reorder=0;          // no reordering of processes
MPI_Comm newcomm;
int nleft, nright, nup, nlow;
int distance=1;      // we are intersted in the direct
                     //neighbors of each process

MPI_Cart_create ( MPI_COMM_WORLD, ndims, periods,
       dims, reorder, &newcomm);
MPI_Cart_shift ( newcomm, 0, distance, &nleft, &nright);
MPI_Cart_shift ( newcomm, 1, distance, &nup, &nlow);
…
// Now you can use nleft, nright etc. for communication
MPI_Send ( buf, cnt, dt, nleft, 0, newcomm);
…
```

MPI_Topo_test

```
MPI_Topo_test( MPI_Comm comm, int *topo_type);
```

- How do I know whether a communicator also has topology information attached to it?
- `topo_type` is one of the following constants:
  - `MPI_CART`: Cartesian topology
  - `MPI_GRAPH`: General graph topology
  - `MPI_UNDEFINED`: no topolgoy, has not been created with MPI_Cart_create (or other, similar functions).

MPI_Dims_create

```
MPI_Dims_create( int np, int ndims, int *dims);
```

- How do I distribute `np` processes best in `ndims` dimensions?
  - `np`: number of process for which to calculate the distribution
  - `ndims`: number of cartesian dimensions
  - `dims`: array containing the extent of each dimension after the call
    - dimensions are set to be as close to each other as possible
    - you can force a certain extent for a dimension by setting its value; only dimensions which are initialized to zero will be calculated

Final example

- Extend the previous example to work for arbitrary number of processes

```
int ndims=2;
int dims[2]= {0,0};       // calculate both dimensions
int periods[2] = {0,0}; // no periodic boundaries
int reorder=0;            // no reordering of processes
MPI_Comm newcomm;
int nleft, nright, nup, nlow;
int distance=1;       // we are interested in the direct
                      //neighbors of each process

MPI_Comm_size ( MPI_COMM_WORLD, &size;)
MPI_Dims_create ( size, ndims, dims );
MPI_Cart_create ( MPI_COMM_WORLD, ndims, dims, periods,
              reorder, &newcomm);
MPI_Cart_shift ( newcomm, 0, distance, &nleft, &nright);
MPI_Cart_shift ( newcomm, 1, distance, &nup, &nlow);
```

What else is there ?

- Creating a communicator, where the processes are ordered logically as described by a directed graph using `MPI_Graph_create`
- Creating a communicator consisting of two process groups
  - also called an inter-communicator
  - local and remote group have however separate ranking scheme
    ➔you have two processes having the rank 0, one in the local group and one in the remote group
- Dynamically adding processes (`MPI_Comm_spawn`)
- Connecting two independent applications
  (`MPI_Comm_connect/MPI_Comm_accept`)