TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

# MPI-3

Victor Eijkhout eijkhout@tacc.utexas.edu

TACC/XSEDE MPI training 2020



TACC

# Code of Conduct

XSEDE has an external code of conduct for XSEDE sponsored events which represents XSEDE's commitment to providing an inclusive and harassment-free environment in all interactions regardless of gender, sexual orientation, disability, physical appearance, race, or religion. The code of conduct extends to all XSEDE-sponsored events, services, and interactions.

Code of Conduct: `https://www.xsede.org/codeofconduct`

Contact:

- Teacher: Victor Eijkhout `eijkhout@tacc.utexas.edu`
- Event organizer: Jason Allison `jasona@tacc.utexas.edu`

XSEDE ombudspersons:

- Linda Akli, Southeastern Universities Research Association `akli@sura.org`
- Lizanne Destefano, Georgia Tech `lizanne.destefano@ceismc.gatech.edu`
- Ken Hackworth, Pittsburgh Supercomputing Center `hackworth@psc.edu`
- Bryan Snead, Texas Advanced Computing Center `jbsnead@tacc.utexas.edu`

# Justification

Version 3 of the MPI standard has added a number of features, some geared purely towards functionality, others with an eye towards efficiency at exascale.

# Part I

## Fortran bindings

# How to use derived types

Create, commit, use, free:

```
MPI_Datatype newtype;
MPI_Type_xxx( ... oldtype ... &newtype);
MPI_Type_commit ( &newtype );

  // code using the new type

MPI_Type_free ( &newtype );
```

```
Type(MPI_Datatype) :: newtype ! F2008
Integer            :: newtype ! F90
```

The `oldtype` can be elementary or derived.
Recursively constructed types.

# MPI definitions

You need an include file:

```
#include "mpi.h" // for C
use mpi        ! for Fortran90
use mpi_f08    ! for Fortran2008
```

- There are no real C++ bindings.
- True Fortran bindings as of the 2008 standard. Provided in Intel compiler:

  `module load intel/18.0.2`

  or newer. Not in gcc7.

# MPI Init / Finalize

Then put these calls around your code:

```
ierr = MPI_Init(&argc,&argv); // zeros allowed
// your code
ierr = MPI_Finalize();
```

and for Fortran:

```
call MPI_Init(ierr) ! F90 style
call MPI_Init()     ! F08 style
! your code
call MPI_Finalize(ierr) ! F90 style
call MPI_Finalize()     ! F08 style
```

# About error codes

MPI routines return an integer error code

- In C: function result. Can be ignored.
- In Fortran: as optional (F08 only) parameter.
- In Python: throwing exception.

There's actually not a lot you can do with an error code:
very hard to recover from errors in parallel.
Just ignore them . . .

# Part II

## Non-blocking collectives

# Non-blocking collectives

- Collectives are blocking.
- Compare blocking/non-blocking sends:

  `MPI_Send` → `MPI_Isend`
- Non-blocking collectives:

  `MPI_Bcast` → `MPI_Ibcast`
- Use for overlap communication/computation
- Imbalance resilience
- Allows pipelining

# Use of non-blocking collectives

- Similar calls, but output a request object:

```
MPI_Isomething( <usual arguments>, MPI_Request *req);
```

- Calls return immediately;
  the usual story about buffer reuse
- Requires **`MPI_Wait...`** for completion.
- Multiple collectives can complete in any order
- No guaranteed progress.

# MPI_Ibcast

C:
```
int MPI_Ibcast(
    void* buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm,
    MPI_Request *request
    )
```

Fortran:
```
MPI_Ibcast(buffer, count, datatype, root, comm, ierror)
TYPE(*), DIMENSION(..) :: buffer
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
TYPE(MPI_Request),intent(out) :: request
```

# Overlapping collectives

Independent collective and local operations:

$$y \leftarrow Ax + (x^t x)y$$

```
MPI_Iallreduce( .... x ..., &request);
// compute the matrix vector product
MPI_Wait(request);
// do the addition
```

# Exercise 1 (`procgridnonblock`)

▸ Earlier procgrid exercise

Revisit exercise 5. Let only the first row and first column have certain data, which they broadcast through columns and rows respectively. Each process is now involved in two simultaneous collectives. Implement this with non-blocking broadcasts, and time the difference between a blocking and a non-blocking solution.

# Non-blocking barrier

# Just what is a barrier?

- Barrier is not *time* synchronization but *state* synchronization.
- Test on non-blocking barrier: 'has everyone reached some state'

# Use case: adaptive refinement

- Some processes decide locally to alter their structure
- . . . need to communicate that to neighbours
- Problem: neighbours don't know whether to expect update calls, if at all.
- Solution: do update calls, if any, then post barrier.
  Everyone probe for updates, test for barrier.

# Use case: distributed termination detection

- Distributed termination detection (Matocha and Kamp, 1998): draw a global conclusion with local operations
- Everyone posts the barrier when done;
- keeps doing local computation while testing for the barrier to complete

# MPI_Ibarrier

```
C:
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)

Input Parameters
comm : communicator (handle)

Output Parameters
request : communication request (handle)

Fortran2008:
MPI_Ibarrier(comm, request, ierror)
Type(MPI_Comm),intent(int) :: comm
TYPE(MPI_Request),intent(out) :: request
INTEGER,intent(out),optional :: ierror
```

```
// ibarriertest.c
for ( ; ; step++) {
  int barrier_done_flag=0;
  MPI_Test(&barrier_request,&barrier_done_flag,
    MPI_STATUS_IGNORE);
  if (barrier_done_flag) {
    break;
  } else {
    int flag; MPI_Status status;
    MPI_Iprobe
      ( MPI_ANY_SOURCE,MPI_ANY_TAG,
        comm, &flag, &status );
  }
}
```

# Exercise 2 (`ibarrierupdate`)

- Let each process send to a random number of randomly chosen neighbours. Use **`MPI_Isend`**.
- Write the main loop with the **`MPI_Test`** call.
- Insert an **`MPI_Iprobe`** call and process incoming messages.
- Can you make sure that all sends are indeed processed?

# Problem with ‘progress’

- Problem: `MPI_Test` is local
- Something needs to force the barrier information to propagate
- Solution: force progress with `MPI_Iprobe`
- Frowny face: barrier completion takes much longer than you’d expect.

# Part III

## Shared memory

# Shared memory myths

Myth:

> *MPI processes use network calls, whereas OpenMP threads access memory directly, therefore OpenMP is more efficient for shared memory.*

Truth:

> *MPI implementations use copy operations when possible, whereas OpenMP has thread overhead, and affinity/coherence problems.*

Main problem with MPI on shared memory: data duplication.

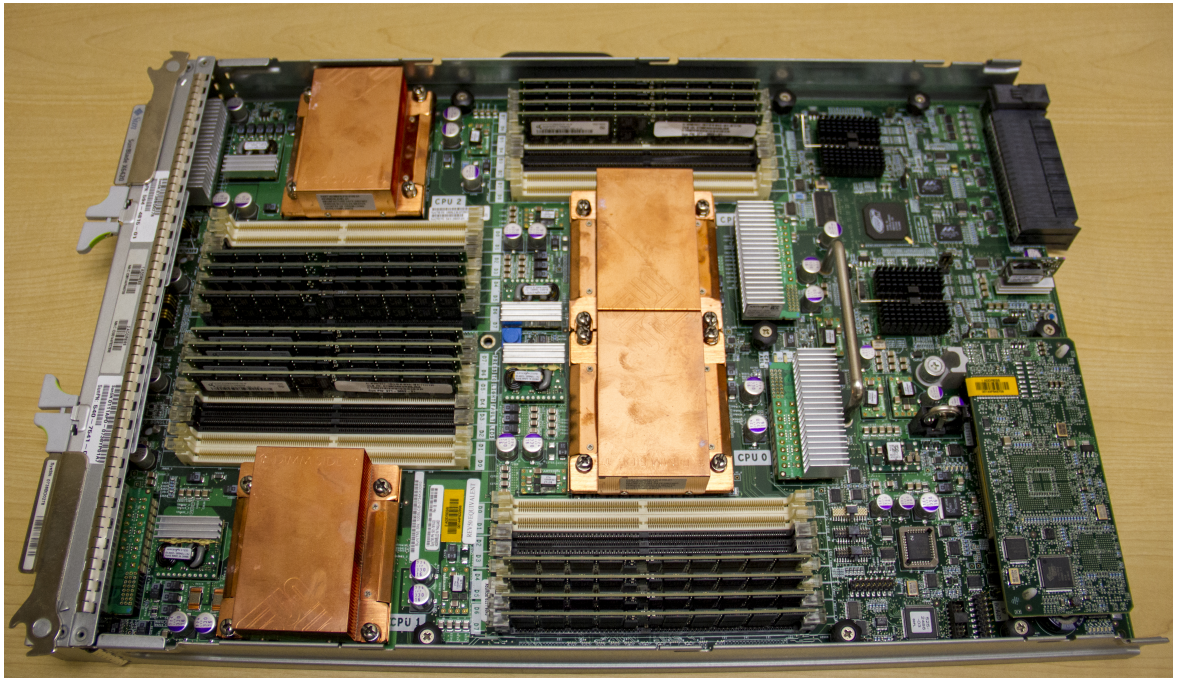
# MPI shared memory

- Shared memory access: two processes can access each other's memory through `double*` (and such) pointers, if they are on the same shared memory.
- Limitation: only window memory.
- Non-use case: remote update. This has all the problems of traditional shared memory (race conditions, consistency).
- Good use case: every process needs access to large read-only dataset Example: ray tracing.

# Shared memory threatments in MPI

- MPI uses optimizations for shared memory: copy instead of socket call
- One-sided offers ‘fake shared memory’: yes, can access another process’ data, but only through function calls.
- MPI-3 shared memory gives you a pointer to another process’ memory, *if that process is on shared memory*.

# Shared memory per cluster node




- Cluster node has shared memory
- Memory is attached to specific socket
- beware Non-Uniform Memory Access (NUMA) effects

# Shared memory interface

Here is the high level overview; details next.

- Use **`MPI_Comm_split_type`** to find processes on the same shared memory
- Use **`MPI_Win_allocate_shared`** to create a window between processes on the same shared memory
- Use **`MPI_Win_shared_query`** to get pointer to another process' window data.
- You can now use `memcpy` instead of **`MPI_Put`**.

# Discover shared memory

- **MPI_Comm_split_type** splits into communicators of same type.
- Only supported type: MPI_COMM_TYPE_SHARED splitting by shared memory.

```
// commsplittype.c
MPI_Info info;
MPI_Comm_split_type(MPI_COMM_WORLD,MPI_COMM_TYPE_SHARED,procno,
    info,&sharedcomm);
MPI_Comm_size(sharedcomm,&new_nprocs);
MPI_Comm_rank(sharedcomm,&new_procno);

ASSERT(new_procno<CORES_PER_NODE);
```

# Allocate shared window

Use **MPI_Win_allocate_shared** to create a window that can be shared;

- Has to be on a communicator on shared memory
- memory will be allocated contiguously
  convenient for address arithmetic,
  not for NUMA: set `alloc_shared_noncontig` true in **MPI_Info** object

```
// sharedbulk.c
MPI_Aint window_size; double *window_data; MPI_Win
    node_window;
if (onnode_procid==0)
  window_size = sizeof(double);
else window_size = 0;
MPI_Win_allocate_shared
  ( window_size,sizeof(double),MPI_INFO_NULL,
    nodecomm,
    &window_data,&node_window);
```

# Get pointer to other windows

Use **MPI_Win_shared_query**:

```
MPI_Aint window_size0; int window_unit; double *win0_addr;
MPI_Win_shared_query( node_window,0,
                      &window_size0,&window_unit, &win0_addr )
    ;
```

# Boring example: heat equation

```
// sharedshared.c
MPI_Win_allocate_shared(3,sizeof(int),info,sharedcomm,&
    shared_baseptr,&shared_window);
```

# Exciting example: bulk data

- Application: ray tracing:
  large read-only data strcture describing the scene
- traditional MPI would duplicate:
  excessive memory demands
- Better: allocate shared data on process 0 of the shared communicator
- every else points to this object.

# Exercise 3 (`shareddata`)

Let the 'shared' data originate on process zero in `MPI_COMM_WORLD`. Then:

- create a communicator per shared memory domain;
- create a communicator for all the processes with number zero on their node;
- broadcast the shared data to the processes zero on each node.

# Part IV

## Process topologies

# Overview

This section discusses graph topologies.

Commands learned:

- **`MPI_Dist_graph_create`**, `MPI_DIST_GRAPH`
- *`MPI_Neighbor_...`*

# Process topologies

- Processes don't communicate at random
- Example: Cartesian grid, each process 4 (or so) neighbours
- Express operations in terms of topology
- Elegance of expression

# Process reordering

- Consecutive process numbering often the best:
  divide array by chunks
- Not optimal for grids or general graphs:
- MPI is allowed to renumbering ranks
- Graph topology gives information from which to deduce renumbering

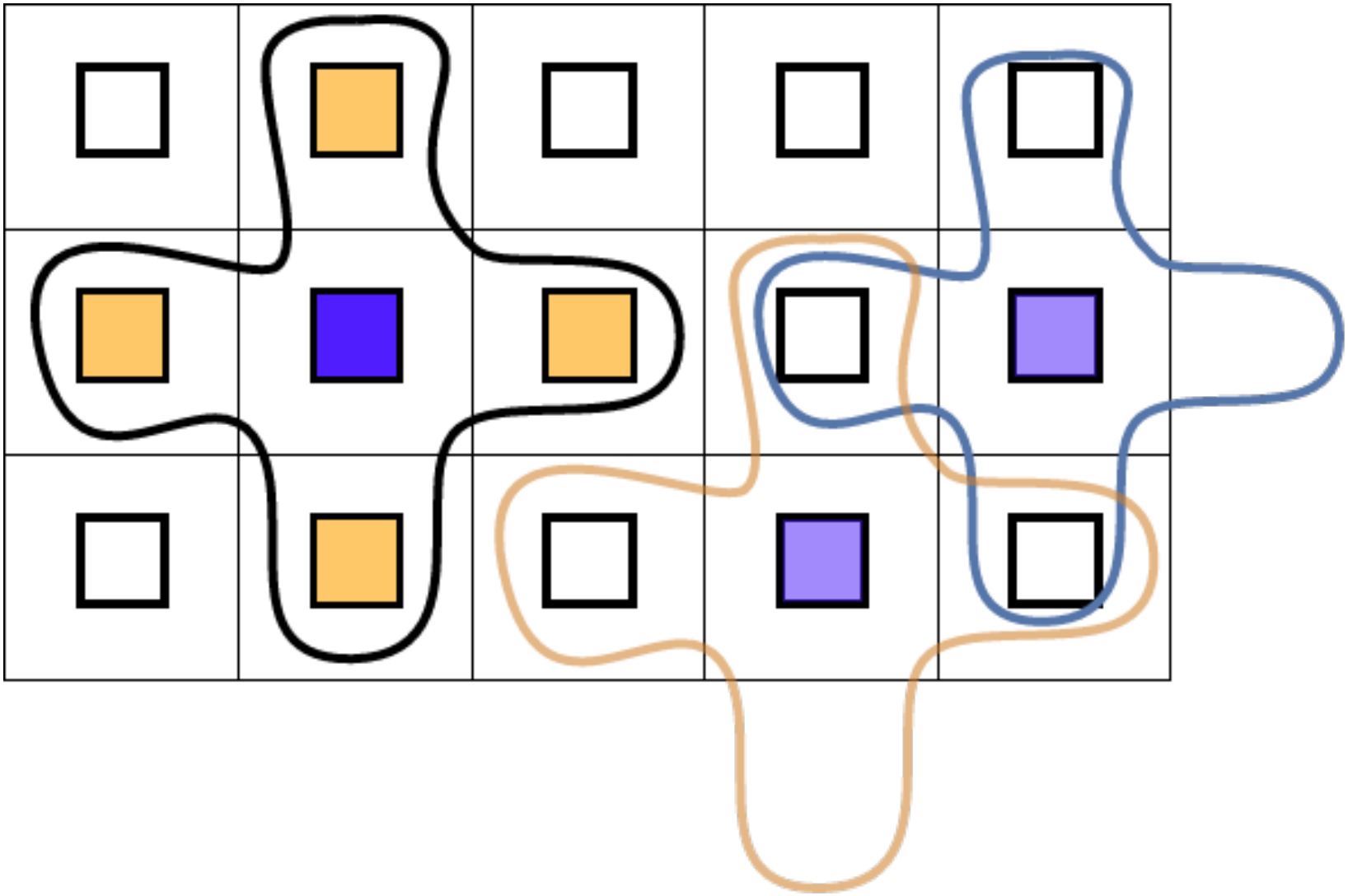
# MPI-1 topology

- Cartesian topology
- Graph topology, globally specified.
  Not exactly scalable!

# MPI-2 and 3 topology

- Graph topologies locally specified: scalable!
- Neighborhood collectives:
  expression close to the algorith.

# Neighbourhood collective



Distributed graph topology where each node has four neighbours

# Whyy neighbourhood collectives?

- Using `MPI_Isend` / `MPI_Irecv` is like spelling out a collective;
- Collectives can use pipelining as opposed to sending a whole buffer;
- Collectives can use spanning trees as opposed to direct connections.

# Create graph topology

```
int MPI_Dist_graph_create
    (MPI_Comm comm_old, int n, const int sources[],
     const int degrees[], const int destinations[], const int
      weights[],
     MPI_Info info, int reorder,
     MPI_Comm *comm_dist_graph)
```

- *nsources* how many processes described? (Usually 1)
- *sources* the processes being described (Usually **MPI_Comm_rank** value)
- *degrees* how many processes to send to
- *destinations* their ranks
- *weights*: usually set to MPI_UNWEIGHTED.
- *info*: MPI_INFO_NULL will do
- *reorder*: 1 if dynamically reorder processes

# Neighbourhood collectives

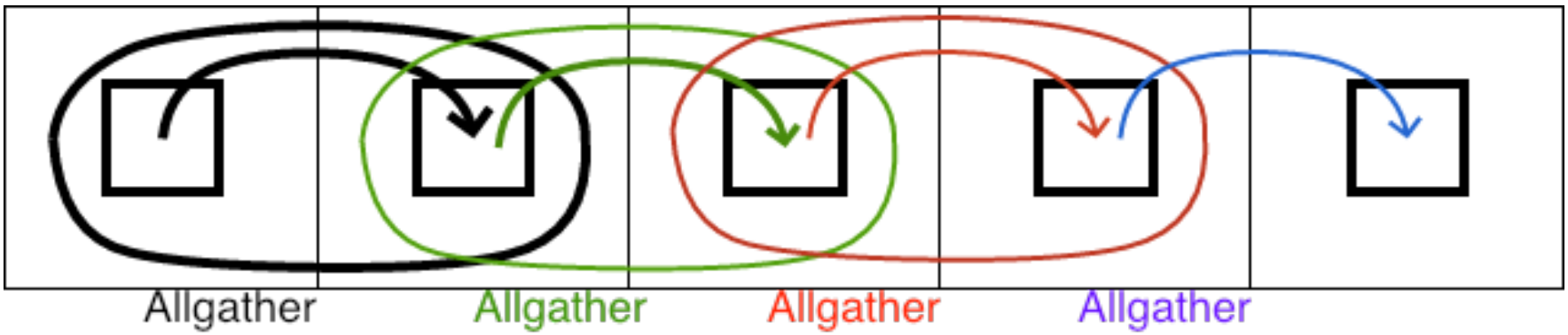
```
int MPI_Neighbor_allgather
   (const void *sendbuf, int sendcount,MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    MPI_Comm comm)
```

Like an ordinary **MPI_Allgather**, but
the receive buffer has a length *degree*.

# Exercise 4 (`rightgraph`)

Revisit exercise **??** and solve it using **`MPI_Dist_graph_create`**. Use figure 46 for inspiration.

# Inspiring picture for the previous exercise




Solving the right-send exercise with neighbourhood collectives

# Part V

# Process management

# Inter-communicators

- Communicators so far are of *intra-communicator* type.
- Bridge between two communicators: *inter-communicator*.
- Example: communicator with newly spawned processes

# In a picture

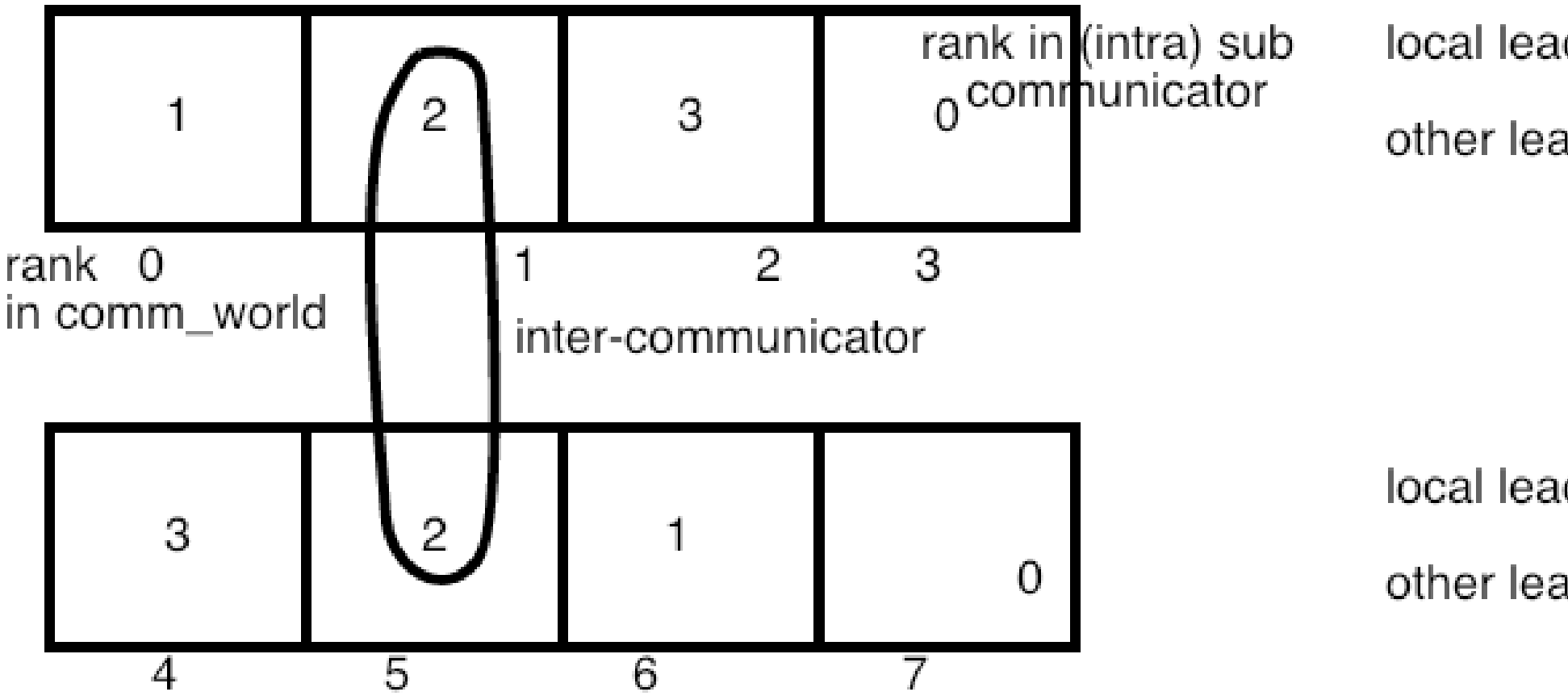



Illustration of ranks in an inter-communicator setup

# Concepts

- Two local communicators
- The ‘peer’ communicator that contains them
- Leaders in each of them
- An inter-communicator over the leaders.

# Routines

- **`MPI_Intercomm_create`**: create
- **`MPI_Comm_get_parent`**: the other leader (see process management)
- **`MPI_Comm_remote_size`**
- **`MPI_Comm_remote_group`**: query the other communicator
- **`MPI_Comm_test_inter`**: is this an inter or intra?

# Overview

This section discusses processes management; intra communicators.

Commands learned:

- **MPI_Comm_spawn**, MPI_UNIVERSE_SIZE
- **MPI_Comm_get_parent**, **MPI_Comm_remote_size**

# Process management

- PVM was a precursor of MPI: could dynamically create new processes.
- It took MPI a while to catch up.
- Use **`MPI_Attr_get`** to retrieve `MPI_UNIVERSE_SIZE` attribute indicating space for creating more processes outside `MPI_COMM_WORLD`.
- New processes have their own `MPI_COMM_WORLD`.
- Communication between the two communicators: ‘inter communicator’ (the old type is ‘intra communicator’)

# Space for processes

Probably a machine dependent component.
Intel MPI at TACC:

```
MY_MPIRUN_OPTIONS="-usize 8" ibrun -np 4 spawn_manager
```

Discover size of the universe:

```
MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
  (void*)&universe_sizep, &flag);
```

# Manager program

```
MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
       (void*)&universe_sizep, &flag);
MPI_Comm everyone; /* intercommunicator */
int nworkers = universe_size-world_size;
MPI_Comm_spawn(worker_program, /* executable */
    MPI_ARGV_NULL, nworkers,
    MPI_INFO_NULL, 0, MPI_COMM_WORLD, &everyone,
    errorcodes);
```

# Worker program

```
MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
MPI_Comm parent;
MPI_Comm_get_parent(&parent);
MPI_Comm_remote_size(parent, &remotesize);
```

# Exercise 5 (`procgrid`)

Organize your processes in a grid, and make subcommunicators for the rows and columns. For this compute the row and column number of each process.

In the row and column communicator, compute the rank. For instance, on a $2 \times 3$ processor grid you should find:

```
Global ranks:  Ranks in row:  Ranks in colum:
  0  1  2       0  1  2        0  0  0
  3  4  5       0  1  2        1  1  1
```

Check that the rank in the row communicator is the column number, and the other way around.

Run your code on different number of processes, for instance a number of rows and columns that is a power of 2, or that is a prime number. This is one occasion where you could use `ibrun -np 9`; normally you would *never* put a processor count on `ibrun`.