HPC Programming

Message Passing Interface (MPI), Part III

Peter-Bernd Otte, 26.11.2019

Introduction MPI

1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Error Handling
5. Groups & Communicators
6. Collective Communication
7. Dealing with I/O
8. MPI Derived Datatypes
9. Common pitfalls and good practice (“need for speed”)

MPI: different communications modes

Recap

	Blocking	Non-Blocking	note
standard send	MPI_Send	MPI_ISend	synchronous or asynchronous send (depending on message size and implementation) uses internal buffer.
synchronous send	MPI_SSend	MPI_ISSend	Only completes when the receive has started
asynchronous (buffered) send	MPI_BSend	MPI_IBSend	Completes after buffer copy (always).
ready send	MPI_RSend	MPI_IRSend	problematic: mandatory to have matching receive already listening. Not discussed in this lecture. Might be fastest solution.

„i” stands for immediate return

	Blocking	Non-Blocking	note
standard receive	MPI_Recv	MPI_IRecv	works for all sending routines.

MPI: P2P communications, Pros and Cons

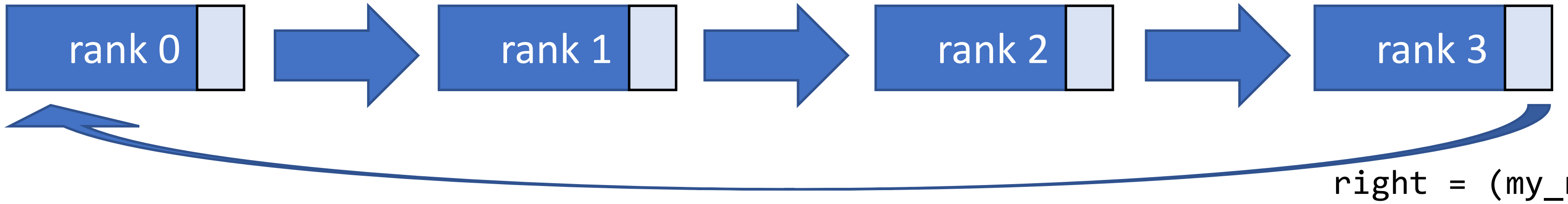
Recap

- synchronous send
  - risk of serialisation, waiting and/or deadlock
  - high latency but best bandwidth
- asynchronous send
  - no risks (except: take care of your buffers)
  - low latency but bad bandwidth
- standard send
  - risk of implementation and message dependence behaviour
  - plus risks of synchronous send

MPI: Non-Blocking Send & Receive

Recap

- to a 1D ring with 1 piece of data passing in one direction




```
right = (my_rank+1) % size;
left = (my_rank-1+size) % size;
```

- cyclic: MPI_Send(…to right…)
  MPI_Recv(…from left…)

deadlock!
All are waiting for a receiver

- non-cyclic: for rank<size-2: MPI_Send(…to right…)
  for rank>0: MPI_Recv(…from left…)

serialisation!
highest rank starts, rank 0 last

(hint: all this only true if MPI calls are synchronous sends)

MPI: Non-Blocking communication

Recap

This can be accomplished by:

- non-blocking send
  1. MPI_Isend();
  2. Different_Work();
  3. MPI_Wait(); //Waits until MPI_Isend completed / send buffer is read out

- non-blocking receive
  1. MPI_Irecv();
  2. Different_Work();
  3. MPI_Wait(); //Waits until MPI_Irecv completed / receive buffer is filled

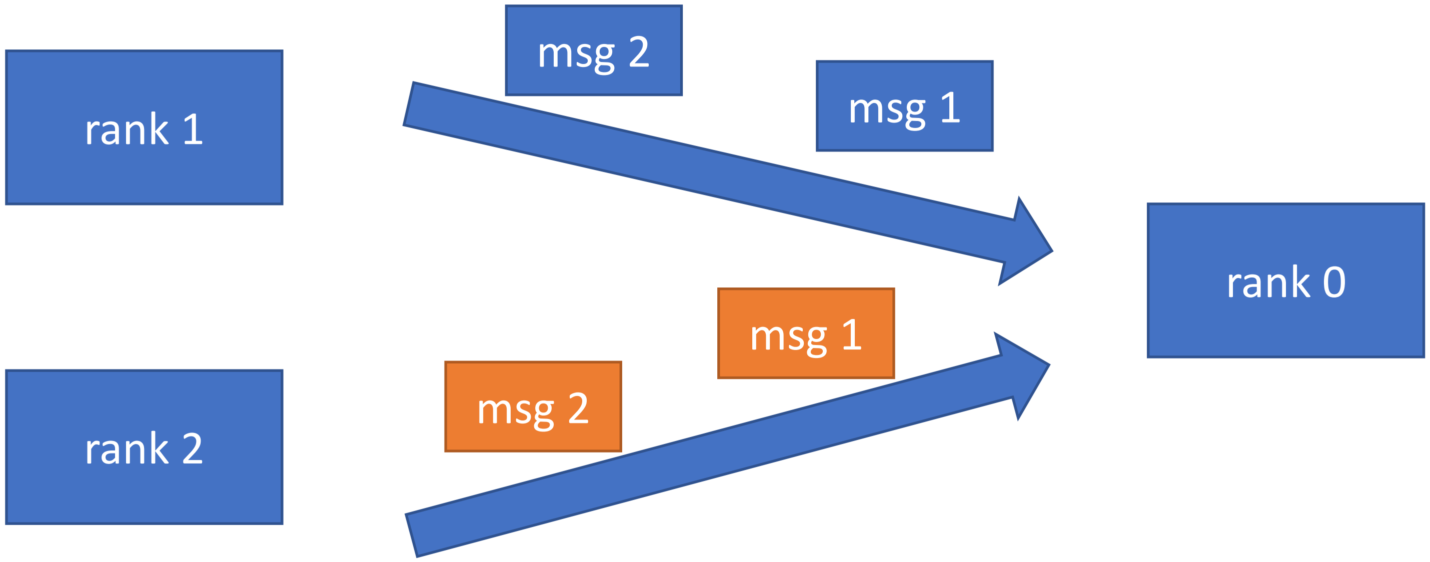
Golden MPI rule:
always <=3 lines of
MPI_* calls per task

otherwise:
check MPI reference or
wrong coding

MPI: Message Order Preservation

Recap

- Messages do not overtake, if same:
  - communicator (eg MPI_COMM_WORLD),
  - source rank and
  - destination rank
- true for: synchronous and asynchronous communications
- messages from different senders can overtake



Introduction MPI

1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Error Handling
5. Groups & Communicators
6. Collective Communication
7. Dealing with I/O
8. MPI Derived Datatypes
9. Common pitfalls and good practice (“need for speed”)

MPI: Error handling

- in short, standard behaviour:
  - MPI: abort on error
  - MPI-IO: continue and just report
  - only if error is detected by MPI, otherwise unpredictable behaviour

- in detail:
  - most important foundation: hardware error free
  - CPU, RAM & network
    - have different techniques to detect hardware errors (eg ECC-RAM, checksums in network packages)
    - you (or your system admin) are informed if hardware problem occurs
  - Change standard behaviour:
    int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
    int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)

Introduction MPI

1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Error Handling
5. Groups & Communicators
6. Collective Communication
7. Dealing with I/O
8. MPI Derived Datatypes
9. Common pitfalls and good practice ("need for speed")

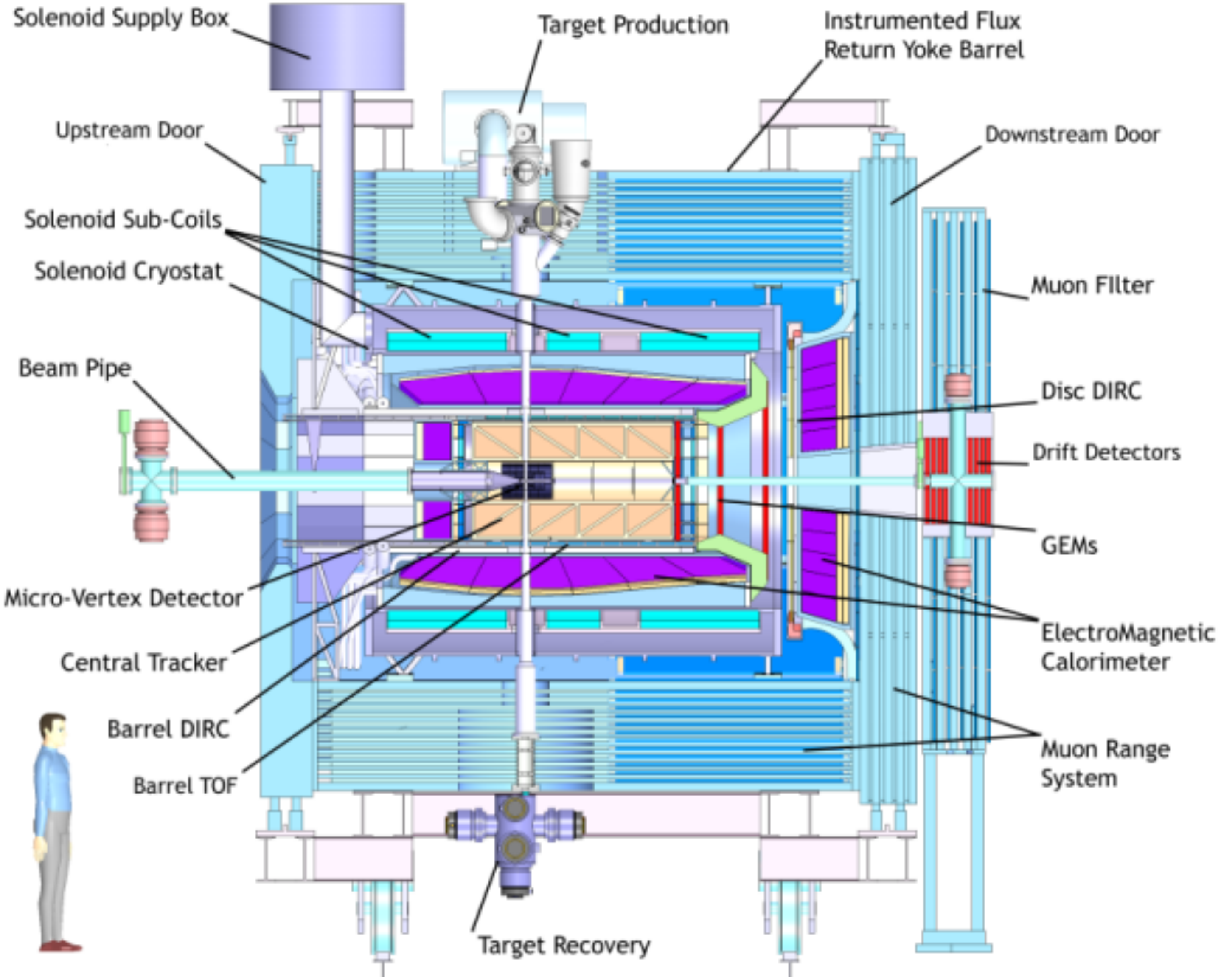
Motivation: Sub-Communicators

- Particle reconstruction
- Multiple layers in detector

- Multiple ranks working in several groups
    - code readability
    - collective communication within group

- OR: a library should NEVER use MPI_COMM_WORLD to not mix up with the main program.

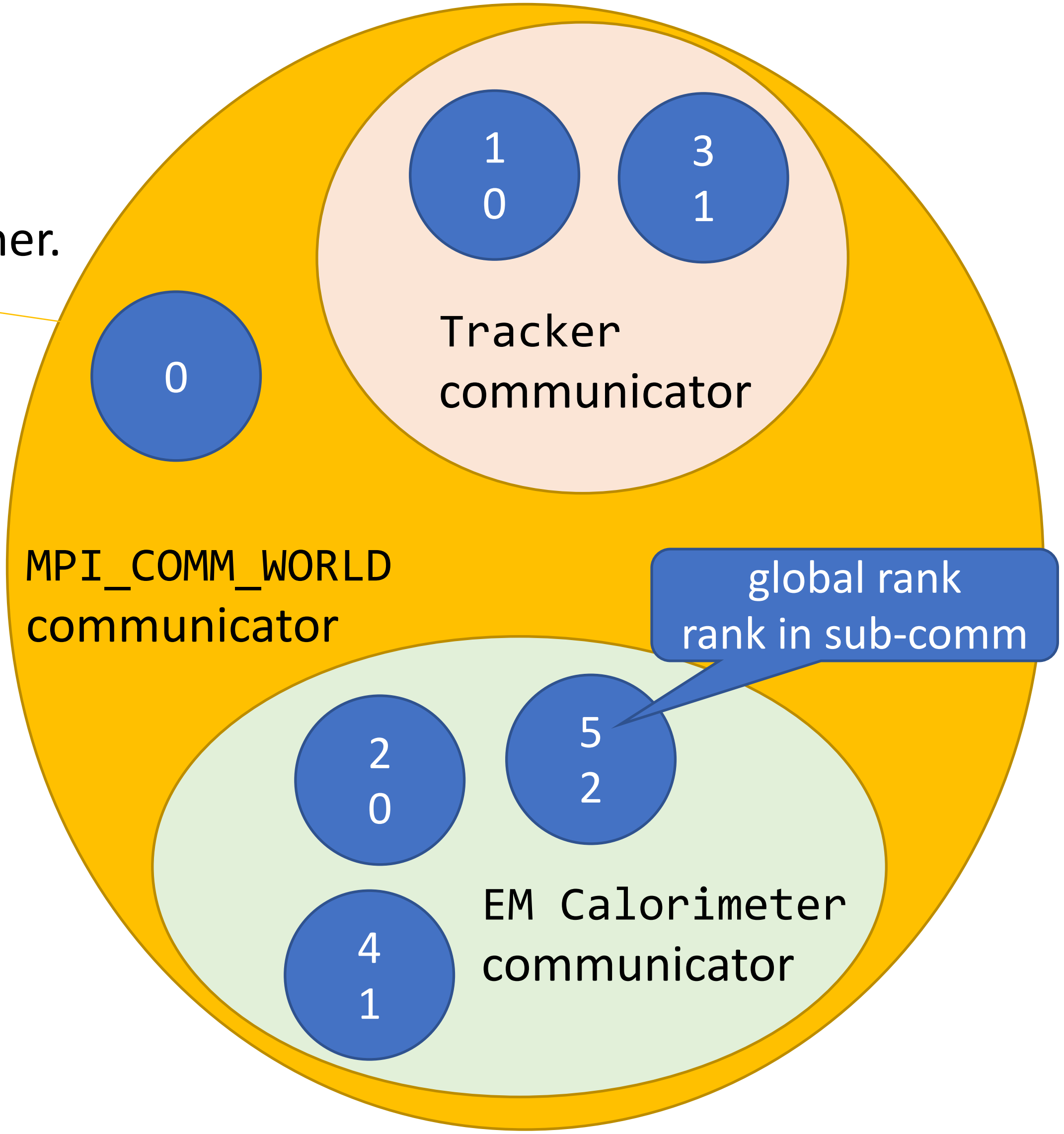
- See Exercise 5



MPI: Sub-Communicators

- MPI Communicator
  = group of processes that can send messages to each other.
- All processes are in `MPI_COMM_WORLD` communicator
- Defining sub groups (eg readability, library):
  1. MPI_Comm_split
  2. MPI_Comm_group + MPI_Comm_create

- Number of members and size in communicator:
  `MPI_Comm_size, MPI_Comm_rank`

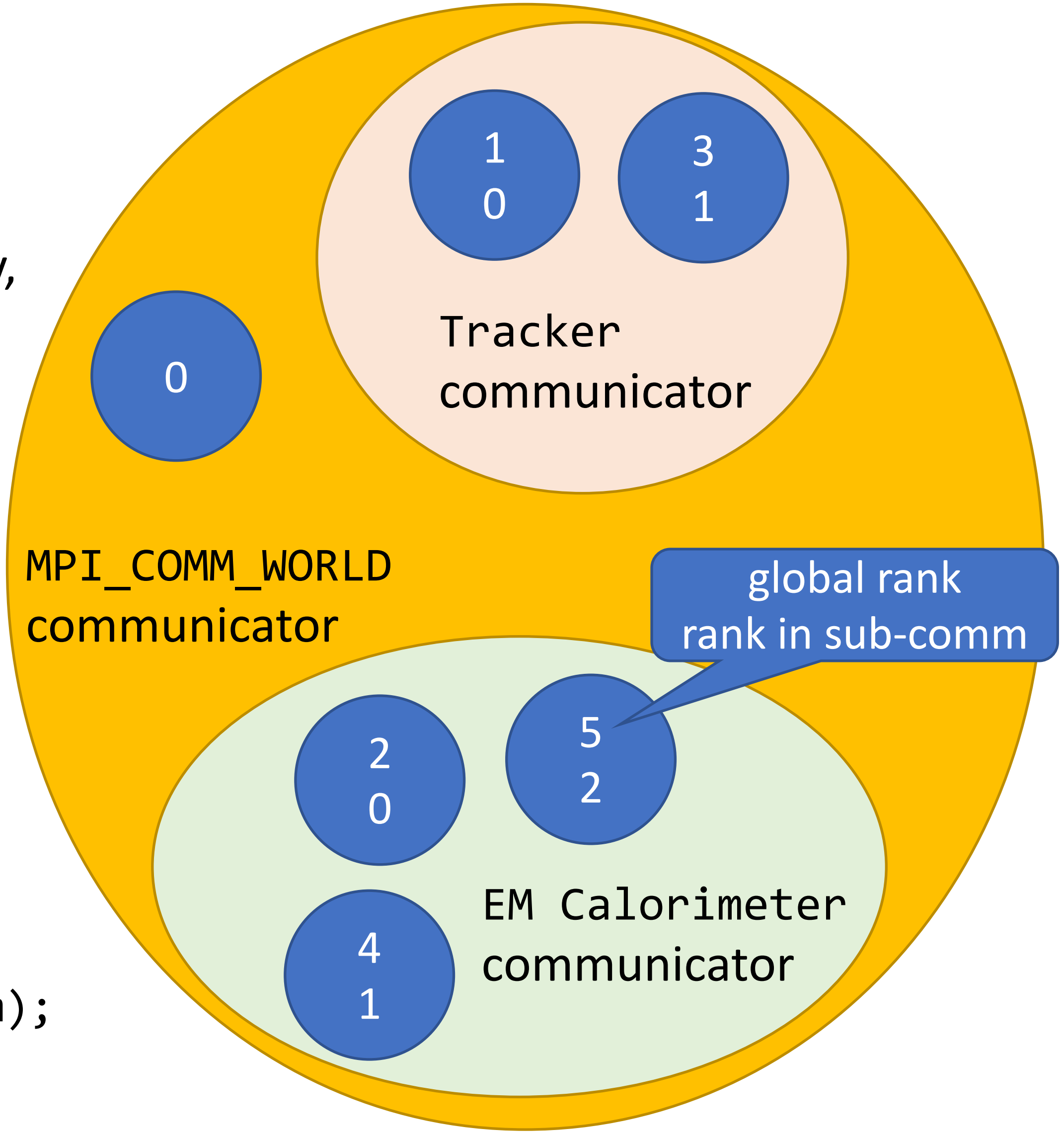


MPI: MPI_Comm_split

- Creates new communicators based on colors
- int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
  - ordering in new group:
    - key == 0 → as sorted in old
    - key != 0 → according to key values
  - one member group: color = MPI_UNDEFINED

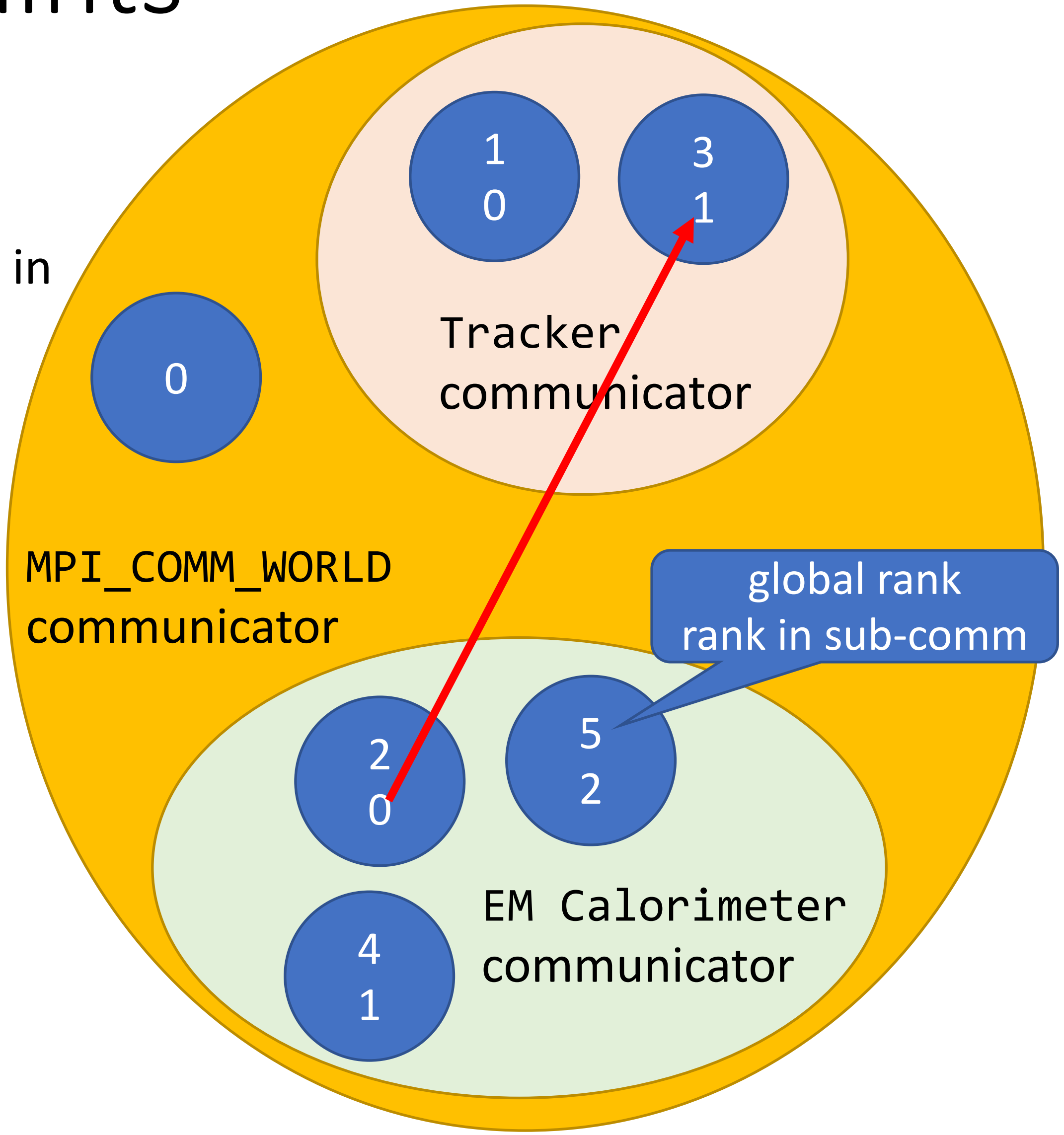
- Example:

```
MPI_Comm newcomm;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
mycolor = my_rank/3;
MPI_Comm_split(MPI_COMM_World, mycolor, 0, &newcomm);
MPI_Comm_rank(newcomm, &my_new_rank);
```



MPI: Sub-Communicators hints

- no difference in speed: same hardware
- Use intra-communicators to communicate between rank in different “worlds” (without MPI_COMM_WORLD ranks)
  - eg MPI_Intercomm_create()

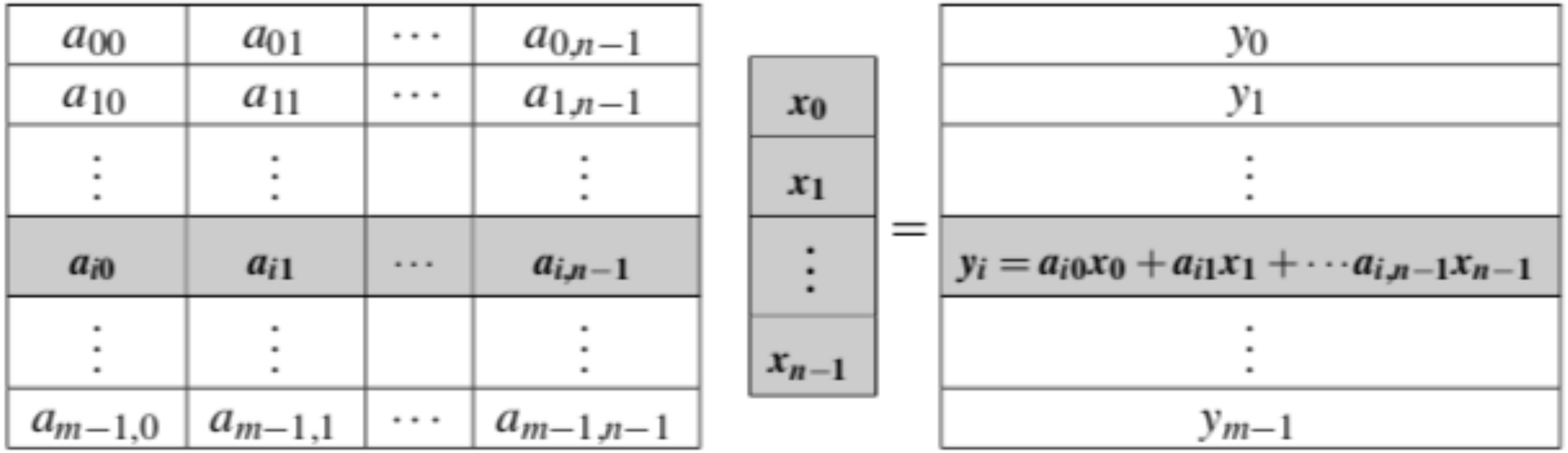


Introduction MPI

1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Error Handling
5. Groups & Communicators
6. Collective Communication
7. Dealing with I/O
8. MPI Derived Datatypes
9. Common pitfalls and good practice (“need for speed”)

Motivation: Collective Communication

- eg matrix multiplication, helpful:
  - reading and spreading of data,
  - gather final results

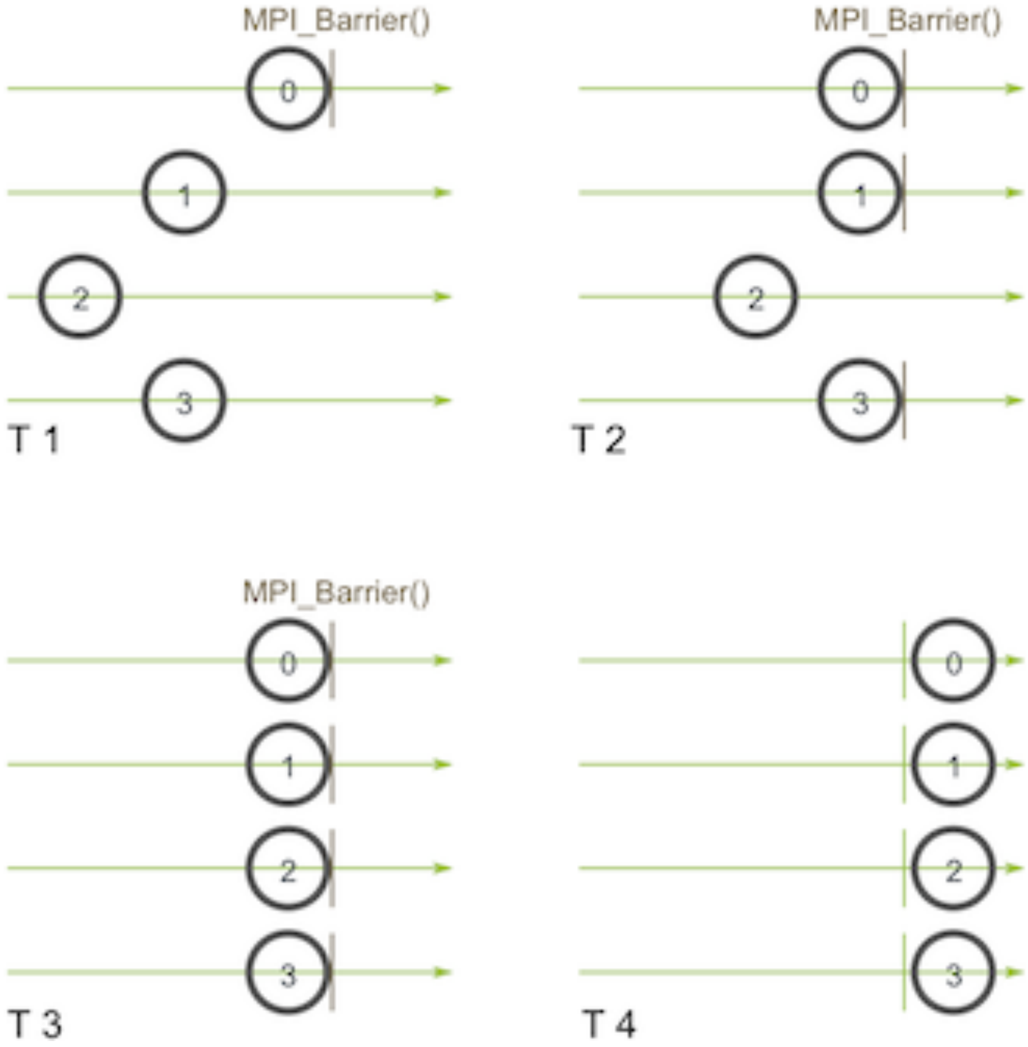


MPI: MPI_Barrier

- collective communication: always include a *synchronization point* among processes.
  - all processes must reach a point in their code before they can all begin executing again.

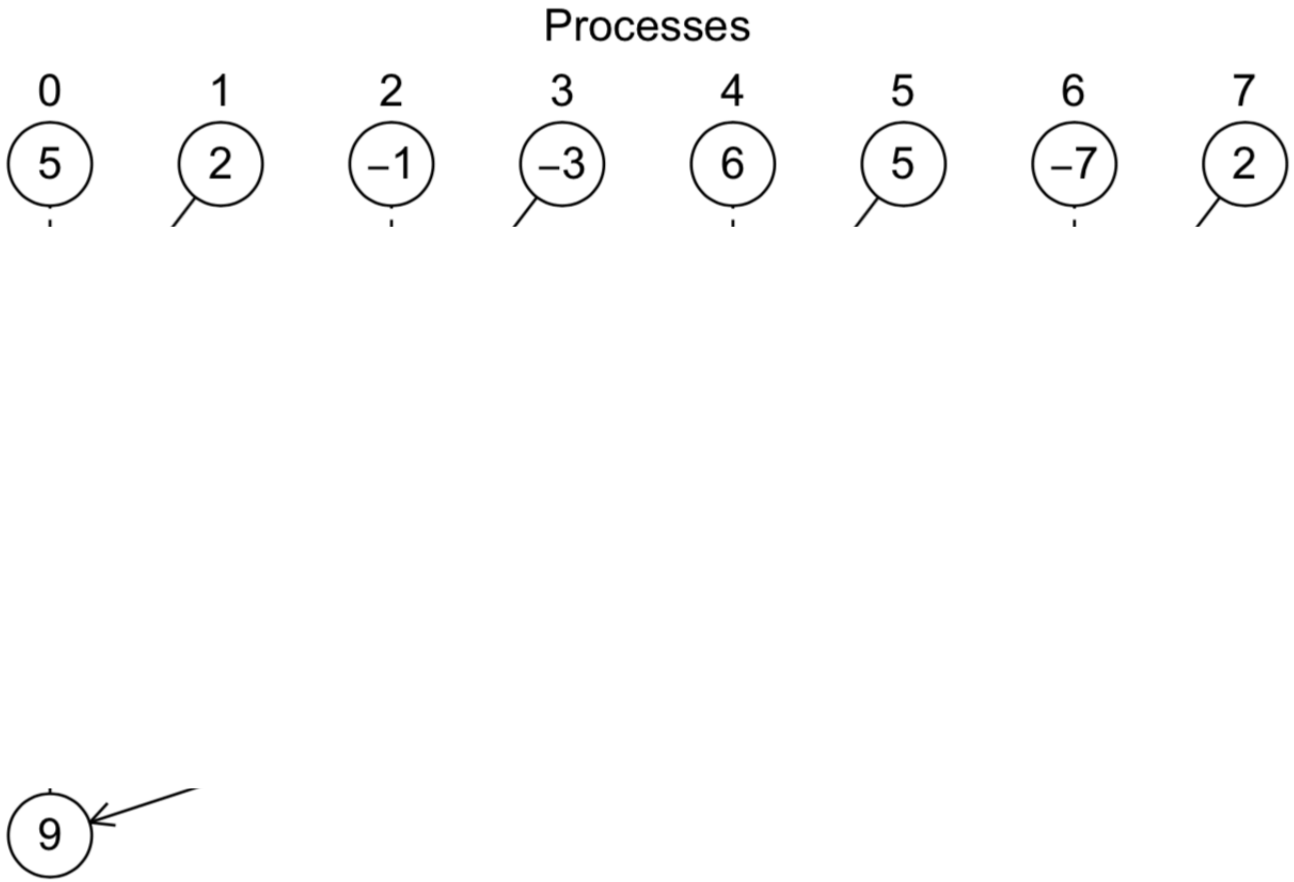
syntax:

MPI_Barrier(MPI_Comm comm);



MPI: MPI_Reduce

- Reduces values on all processes to a single value
  (eg global sum)




```
int MPI_Reduce(
void *sendbuf /*in*/,
void *recvbuf /*out*/,
int count /*in*/,
MPI_Datatype datatype /*in*/,
MPI_Op operator /*in*/,
int dest_process /*in*/,
MPI_Comm comm /*in*/)
```

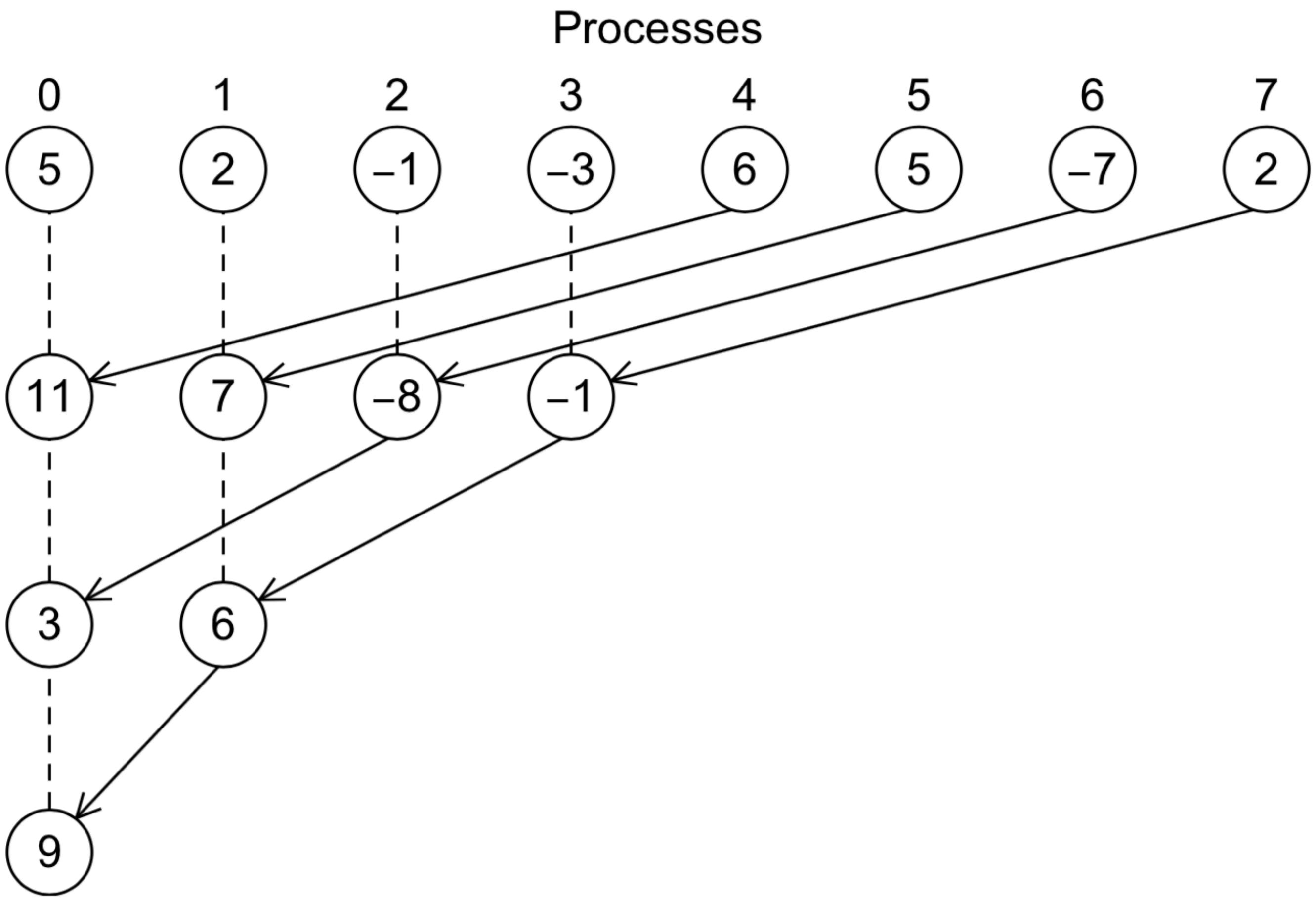
- hints:
  - with count>1, MPI can operate on arrays
  - sendbuf and recvbuf need to different
    (no aliasing!)

MPI: MPI_Reduce

- steps = $\lceil \log_2(N) \rceil$

```
int MPI_Reduce(
void *sendbuf /*in*/,
void *recvbuf /*out*/,
int count /*in*/,
MPI_Datatype datatype /*in*/,
MPI_Op operator /*in*/,
int dest_process /*in*/,
MPI_Comm comm /*in*/)
```

- hint:
  - with count>1, MPI can operate on arrays
  - sendbuf and recvbuf need to different (no aliasing!)



MPI: MPI_Reduce

Worked out example:

```
int local_n, n;

local_n = my_rank;

MPI_Reduce(&local_n /*send_buf*/, &n /*recv_buf*/, 1 /*count*/, MPI_INT,
   MPI_SUM, 0 /*dest_process*/, MPI_COMM_WORLD);
printf("sum of all local_n: %f", n);
```

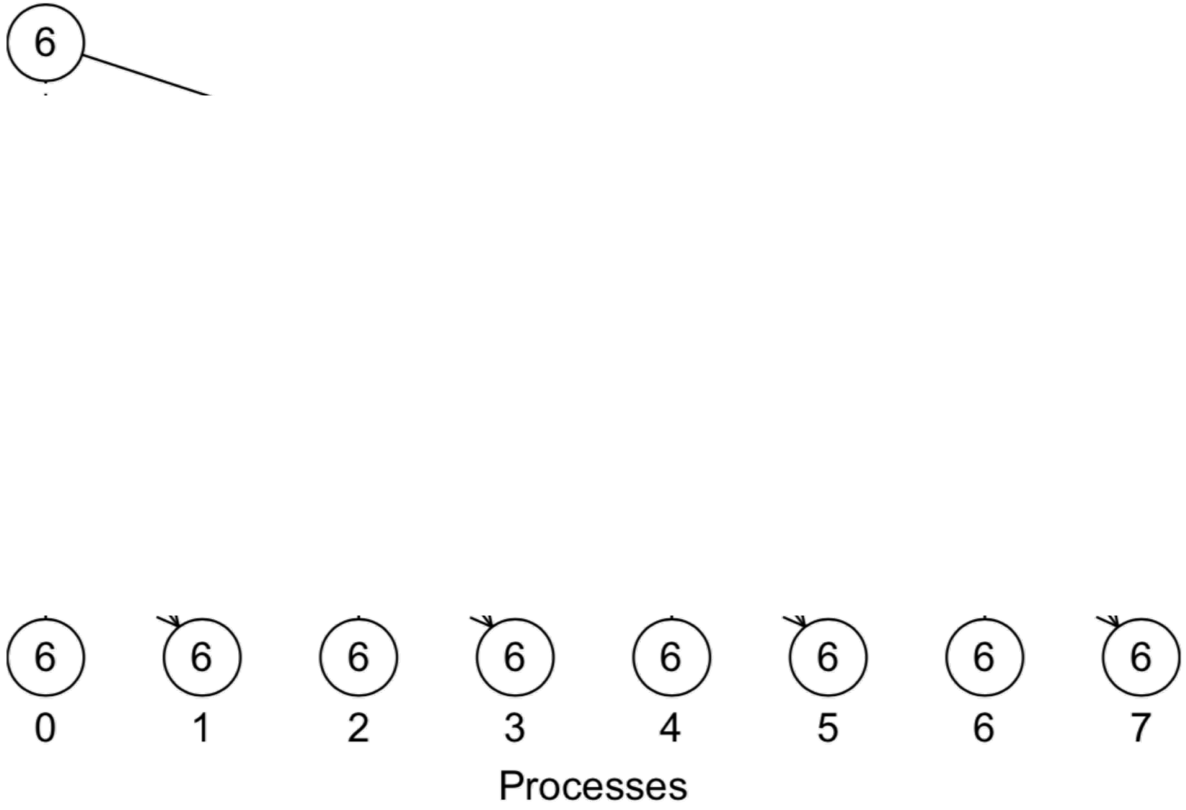
MPI: Reduction Operators

Operation	Meaning
MPI_MAX	Returns the maximum element.
MPI_MIN	Returns the minimum element.
MPI_SUM	Sums the elements.
MPI_PROD	Multiplies all elements.
MPI_LAND	Performs a logical and across the elements.
MPI_LOR	Performs a logical or across the elements.
MPI_BAND	Performs a bitwise and across the bits of the elements.
MPI_BOR	Performs a bitwise or across the bits of the elements.
MPI_MAXLOC	Returns the maximum value and the rank of the process that owns it.
MPI_MINLOC	Returns the minimum value and the rank of the process that owns it.

MPI: P2P ⇔ Collective Communication

- ALL processes in communicator must call SAME collective function at the same time.
- Arguments in all ranks must fit:
  - eg. same dest_process, datatype, operator, comm
  - depending on function
- Only rank dest_process may use recvbuf (but all ranks have to provide such argument)
- MPI_Reduce calls matched solely on:
  - the communicator and
  - the order on which they are called.
  - No helping tags or sender id available.

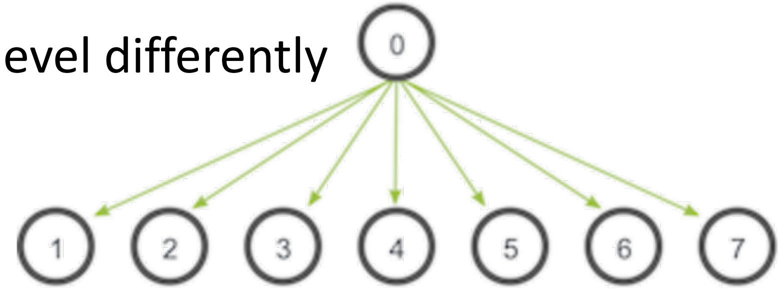
MPI: Broadcast



Broadcasts the same message from the process ”sending_rank” to all other processes of the communicator

- MPI_Bcast(
  void *data,
  int count,
  MPI_Datatype datatype,
  int sending_rank,
  MPI_Comm comm)

- Hint: All ranks have to call this function
- Might be implemented on hardware level differently (MPI implementation should know)



MPI: Broadcast

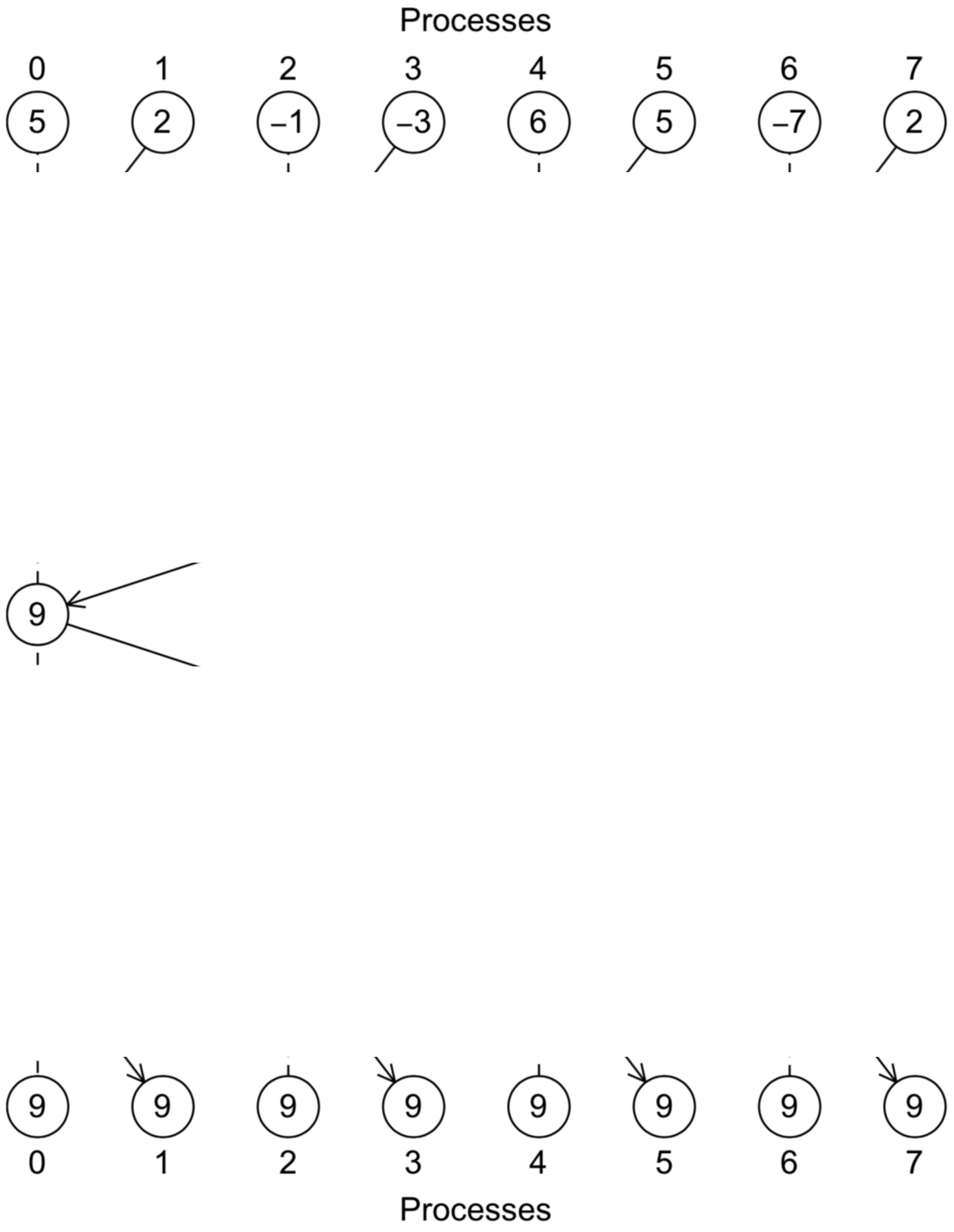
Worked out example, sending 2 variables:

```
int my_rank, my_size, n;
double a;

if (!my_rank) {
        printf(”Enter a and n:\n“);
        scanf(“%lf %d”, &a, &n);
}
MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

MPI: MPI_Allreduce

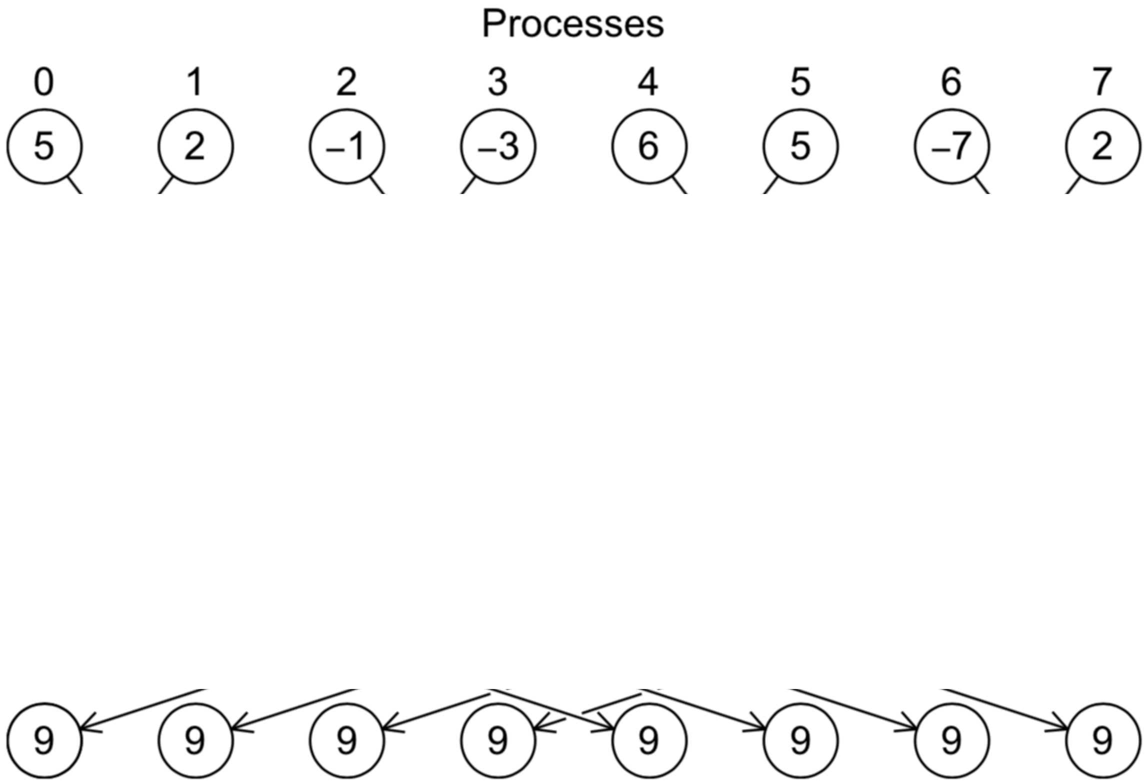
- Combines values from all processes and distributes the result back to all processes
  eg: all processes need global sum
  1. compute global sum (MPI_Reduce) + Broadcast



MPI: MPI_Allreduce

- Combines values from all processes and distributes the result back to all processes
  eg: all processes need global sum
    1. compute global sum (MPI_Reduce) + Broadcast
    2. exchange partial sums (better!, “butterfly”)
- MPI has optimal performance with

```
int MPI_Allreduce(
void *sendbuf /*in*/,
void *recvbuf /*out*/,
int count /*in*/,
MPI_Datatype datatype /*in*/,
MPI_Op operator /*in*/,
MPI_Comm comm /*in*/)
```



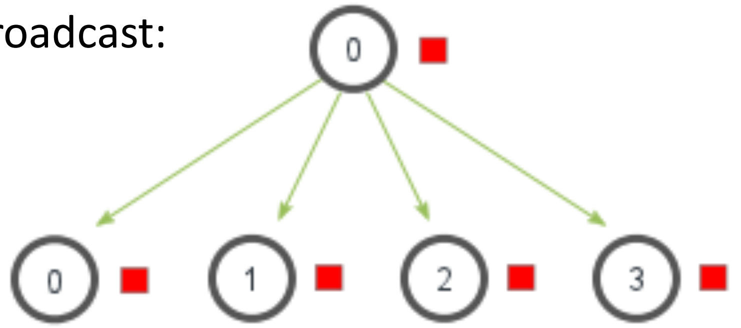
MPI: MPI_Scatter

Sends data from one process to all other processes in a communicator

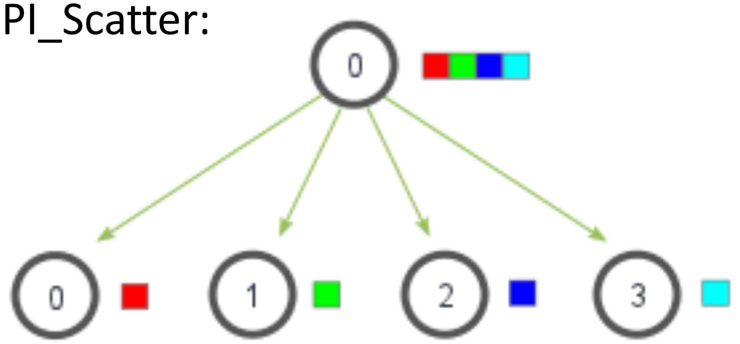
- Selective distribution of data to processes
- MPI_Scatter(
  void* send_data /*in*/,
  int send_count /*in*/,
  MPI_Datatype send_datatype /*in*/,
  void* recv_data /*out*/,
  int recv_count /*in*/,
  MPI_Datatype recv_datatype /*in*/,
  int src_proc /*in*/,
  MPI_Comm comm /*in*/)

- Special cases: MPI_Scatterv

Broadcast:

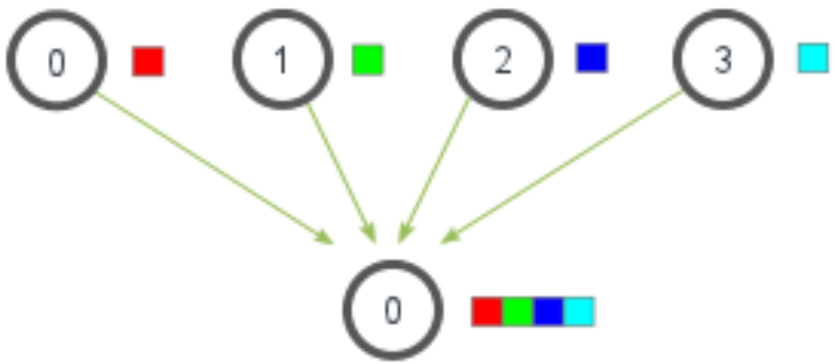


MPI_Scatter:



MPI: MPI_Gather

Gathers together values from a group of processes



- MPI_Gather(
  void* send_data /*in*/,
  int send_count /*in*/,
  MPI_Datatype send_datatype /*in*/,
  void* recv_data /*out*/,
  int recv_count /*in*/,
  MPI_Datatype recv_datatype /*in*/,
  int dest_proc /*in*/,
  MPI_Comm comm /*in*/)

- Special cases: MPI_Gatherv

Exercises 5 and 6

- See online:
  - https://gitlab.rlp.net/pbotte/learnhpc/tree/master/mpi