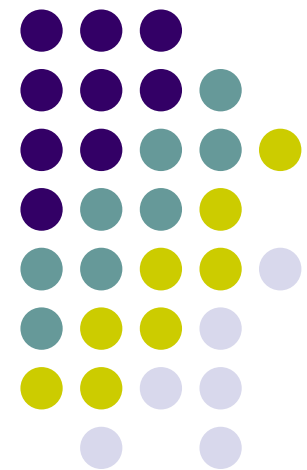


ECE/ME/EMA/CS 759

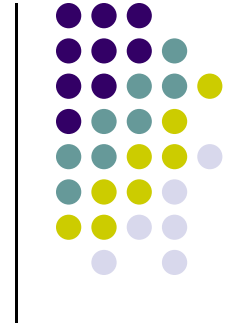
High Performance Computing for Engineering Applications

Parallel Computing via MPI
Non-blocking Point-to-Point
Collective communication

November 13, 2015
Lecture 26



Quote of the Day



“A good traveler has no fixed plans and is not intent on arriving.”

-- Lao-Tzu, Chinese Philosopher

6th–5th Century BC [Zhou Dynasty] - 5th-4th Century BC [Zhou Dynasty]



Before We Get Started

- Issues covered last time:
 - Examples, MPI-enabled parallel computing
 - Point-to-point message passing
 - Blocking message passing
- Today's topics
 - Point-to-point message passing
 - Non-blocking message passing
 - Collective operations
 - Other parallel computing options
- Other issues:
 - HW09, due on Wd, Nov. 18, at 11:59 PM
 - Final Project proposal due tonight in Learn@UW
 - Please call your team a name so that I can use simple names for scheduling/grading/feedback (instead of using four complex Indian names to identify a team)
 - Second (and last) exam: coming up on 11/23 at 7:15 PM (Room TBA)
 - Review on 11/23 in 1610EH during regular lecture hours

Blocking Type: Communication Modes



- Send communication modes:
 - Synchronous send → `MPI_SSEND`
 - Buffered [asynchronous] send → `MPI_BSEND`
 - Standard send → `MPI_SEND`
 - Ready send → `MPI_RSEND`
- Receiving all modes → `MPI_RECV`



1) Synchronous Sending in MPI

2) Buffered Sending in MPI

- Synchronous with MPI_Ssend
 - In synchronous mode, a send will not complete until a matching receive is posted.
 - The sender has to wait for a receive to be posted
 - No buffering of data
 - Used for ensuring the code is healthy and doesn't rely on buffering
- Buffered with MPI_Bsend
 - Send completes once message has been buffered internally by MPI
 - Buffering incurs an extra memory copy
 - Does not require a matching receive to be posted
 - May cause buffer overflow if many bsend's and no matching receives have been posted yet

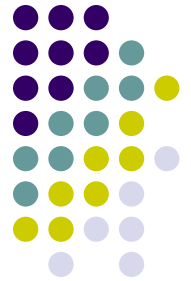


3) Standard Sending in MPI

4) Ready Sending in MPI

- Standard with MPI_Send
 - Up to the MPI implementation to decide whether to do rendezvous or eager, for performance reasons
 - NOTE: If it does rendezvous, in fact the behavior is that of MPI_SSend
 - Very commonly used
- Ready with MPI_Rsend
 - Will work correctly *only* if the matching receive has been posted
 - Can be used to avoid handshake overhead when program is known to meet this condition
 - Rarely used, can cause major problems

Cheat Sheet, Blocking Options



Sender modes	Definition	Notes
Synchronous send MPI_SSEND	Only completes when the receive has started	
Buffered send MPI_BSEND	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Classic MPI_SEND	Standard send	Rendezvous or eager mode. Decided at run time
Ready send MPI_RSEND	Started right away. Will work out only if the matching receive is already posted!	Blindly do a send. Avoid, might cause unforeseen problems...
Receive MPI_RECV	Completes when a the message (data) has arrived	



Looming Danger: Deadlock

- Deadlock situations: appear when due to a certain sequence of commands the execution hangs

```
... Process 0
MPI_Ssend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
```

~~Deadlock~~

No
Deadlock

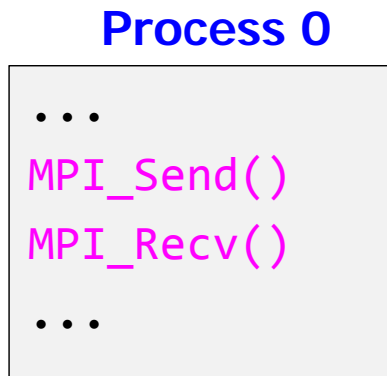
No
Deadlock

```
... Process 1
MPI_Ssend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
...
MPI_Ssend()
MPI_Recv()
...
```




Deadlocking, Another Example

- `MPI_Send` can respond in eager or rendezvous mode
- Example, on a certain machine running MPICH v1.2.1:

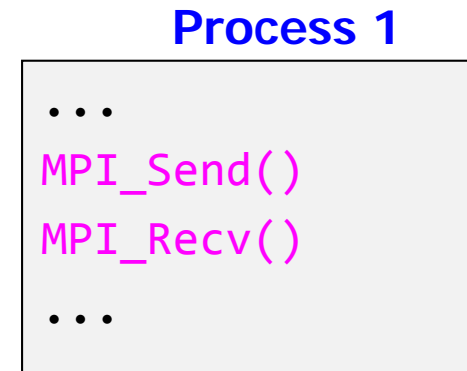


~~Deadlock~~

Data size > 127999 bytes

Data size < 128000 bytes

No
Deadlock





Avoiding Deadlocking

- Easy way to eliminate deadlock is to pair `MPI_Ssend` and `MPI_Recv` operations the right way:

Process 0

```
...  
MPI_Ssend()  
MPI_Recv()  
...
```



Process 1

```
...  
MPI_Recv()  
MPI_Ssend()  
...
```

- Conclusion: understand how the implementation works and what its pitfalls/limitations are

Example



- Always succeeds, even if no buffering is done

```
if(rank==0)
{
    MPI_Send(...);
    MPI_Recv(...);
}
else if(rank==1)
{
    MPI_Recv(...);
    MPI_Send(...);
}
```



Example

- Will always deadlock, no matter the buffering mode

```
if(rank==0)
{
    MPI_Recv(...);
    MPI_Send(...);
}
else if(rank==1)
{
    MPI_Recv(...);
    MPI_Send(...);
}
```



Example

- Only succeeds if message in at least one of the transactions is small enough and an “eager” mode is triggered

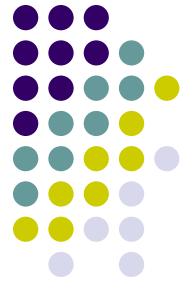
```
if(rank==0)
{
    MPI_Send(...);
    MPI_Recv(...);
}
else if(rank==1)
{
    MPI_Send(...);
    MPI_Recv(...);
}
```



Concluding Remarks, Blocking Options

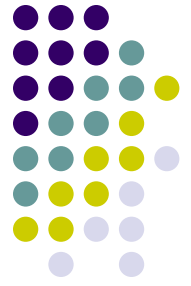
- Standard send (**MPI_SEND**)
 - minimal transfer time if lands in eager mode
 - switches to synchronous mode if in rendezvous mode
- Synchronous send (**MPI_SSEND**)
 - risk of deadlock
 - risk of waiting → idle time
 - high latency / best bandwidth
 - used mostly for debugging purposes
- Buffered send (**MPI_BSEND**)
 - low latency / bad bandwidth
- Ready send (**MPI_RSEND**)
 - use **never**, except you have a 200% guarantee that Recv is already called in the current version and all future versions of your code

Technicalities, Loose Ends: More on the Buffered Send [1/2]



- Relies on the existence of a buffer, which is set up through a call
`int MPI_Buffer_attach(void* buffer, int size);`
- A bsend is a local operation. It does not depend on the occurrence of a matching receive in order to complete
- If a bsend operation is started and no matching receive is posted, the outgoing message is buffered to allow the send call to complete
- Return from an `MPI_Bsend` does not guarantee the message was sent
- Message remains in the buffer until a matching receive is posted

Technicalities, Loose Ends: More on the Buffered Send [2/2]

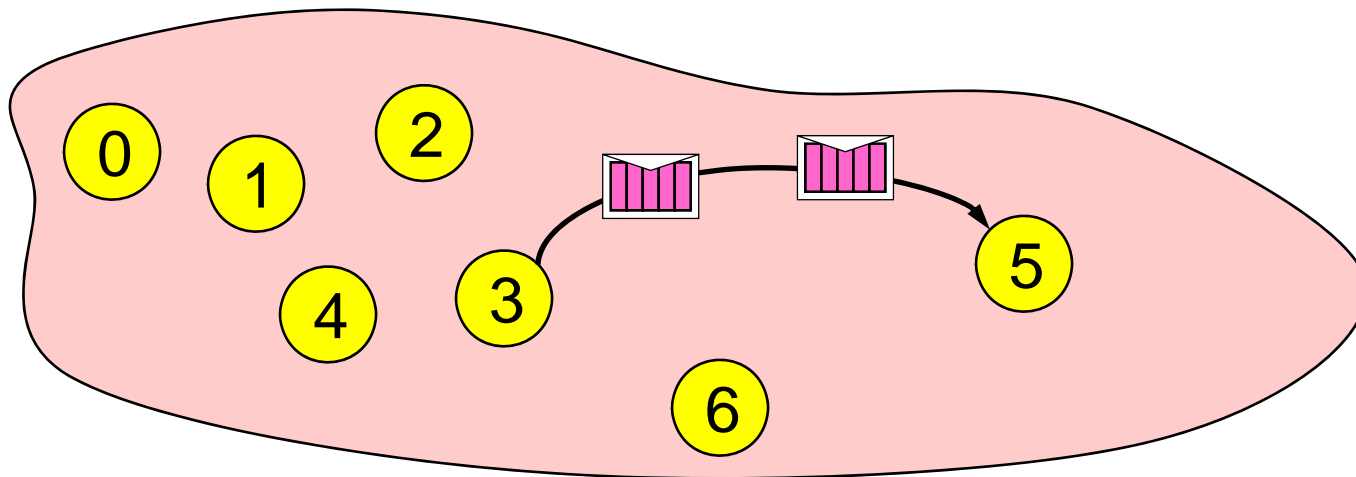


- Ensure you have enough buffer space available.
 - An error occurs if the message must be buffered and there is not enough buffer space
- Amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus $(\text{MPI_BSEND_OVERHEAD} \times \text{number_of_messages})$ should be sufficient
 - `MPI_BSEND_OVERHEAD` – provided by implementation, upper bound on overhead charged by the implementation
- `MPI_Bsend` lowers bandwidth since it requires an extra memory-to-memory copy of the outgoing data
- The `MPI_Buffer_attach` call provides MPI a buffer in the user's memory.
 - Buffer used only by messages sent in buffered mode,
 - Only one buffer attached to a process at any time

Technicalities, Loose Ends: Message Order Preservation

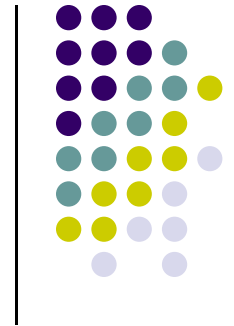


- Rule for messages on the same connection; i.e., same communicator, source, and destination rank:
 - **Messages do not overtake each other**
 - True even for non-synchronous sends

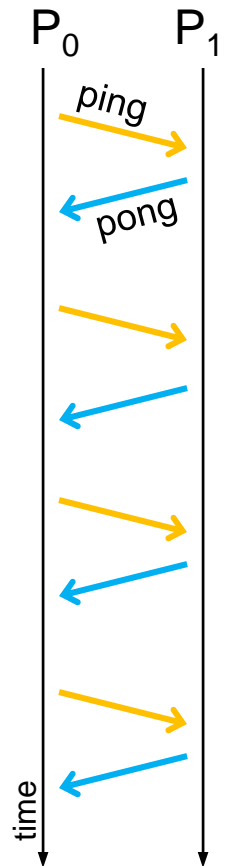


- If both receives match both messages, then the order is preserved

Read This for Assignment 11

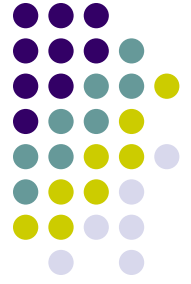


- Write a program according to the time-line diagram:
 - process 0 sends a message to process 1 (ping)
 - after receiving this message, process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Add timing calls before and after the loop
- For timing purposes, you might want to use
`double MPI_Wtime();`
- `MPI_Wtime` returns a wall-clock time in seconds
- At process 0, print out the transfer time in seconds
 - Might want to use a log scale



More on Timing

[Useful, for Assignment 11]

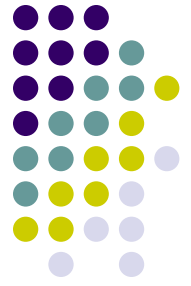


```
int main()
{
    double starttime, endtime;
    MPI_Barrier(MPI_COMM_WORLD);
    starttime = MPI_Wtime();
    .... here comes stuff to be timed ...
    MPI_Barrier(MPI_COMM_WORLD);
    endtime   = MPI_Wtime();
    printf("That took %f seconds\n", endtime - starttime);
    return 0;
}
```

- Resolution is typically 1E-3 seconds
- Time of different processes might actually be synchronized, controlled by the variable `MPI_WTIME_IS_GLOBAL`

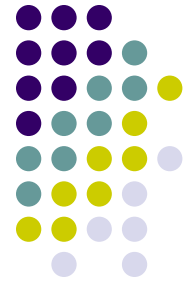
More on Timing

[Useful, for Assignment 11; Cntd.]



- Latency = transfer time for zero length messages
- Bandwidth = message size (in bytes) / transfer time

- Message transfer time and bandwidth change based on the nature of the MPI send operation
 - Standard send (`MPI_Send`)
 - Synchronous send (`MPI_Ssend`)
 - Buffered send (`MPI_Bsend`)
 - Etc.



Non-Blocking Communication

Non-Blocking Communications: Motivation



- Overlap communication with execution (just like w/ CUDA):
 - Phase 1: Initiate non-blocking communication
 - Returns **I**mmediately
 - Routine name starting with MPI_**I**...
 - Phase 2: Do some work
 - “latency hiding”
 - Phase 3: Wait for non-blocking communication to complete



Non-blocking Send/Receive

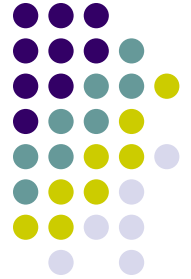
- Syntax

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request);
```

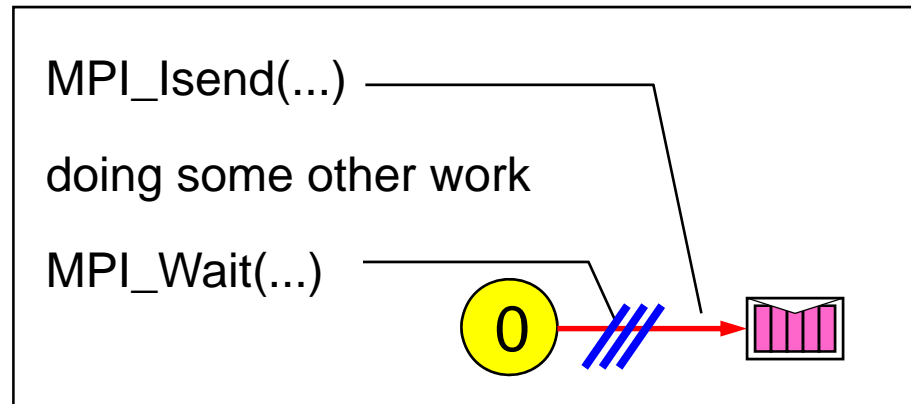
- buf - [in] initial address of send buffer (choice)
- count - [in] number of elements in send buffer (integer)
- datatype - [in] datatype of each send buffer element (handle)
- dest - [in] rank of destination (integer)
- tag - [in] message tag (integer)
- comm - [in] communicator (handle)
- request - [out] communication request (handle)

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Request *request);
```

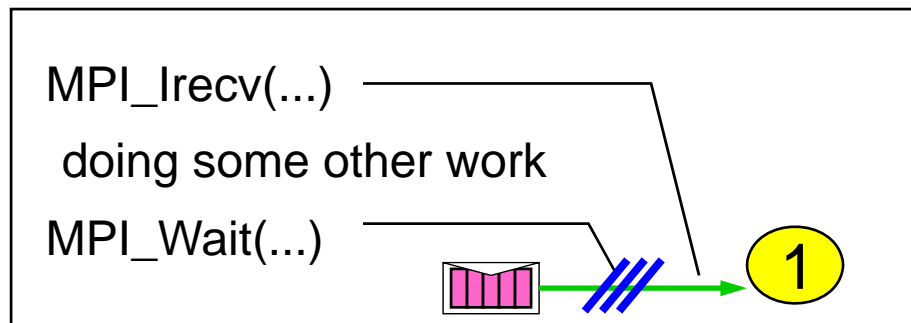
The Screenplay: Non-Blocking P2P Communication



- Non-blocking send



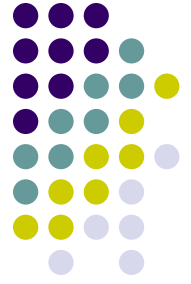
- Non-blocking receive



 = waiting until operation locally completed

Non-Blocking Send/Receive

Some Tools of the Trade



- Call returns immediately. Therefore, user must worry whether ...
 - At the “send” end, data to be sent is out of the send buffer before trampling on the buffer
 - At the “receive” end, data to be received has finished arriving before using the content of the buffer
- Tools that come in handy:
 - For sends and receives in flight
 - `MPI_Wait` – blocking - you go synchronous
 - `MPI_Test` – non-blocking - returns quickly with status information
 - Check for existence of data to receive
 - Blocking: `MPI_Probe`
 - Non-blocking: `MPI_Iprobe`

Waiting for isend/ireceive to Complete



- Waiting on a single send

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

- Waiting on multiple sends (get status of all)

- Till all complete, as a barrier

```
int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses);
```

- Till at least one completes

```
int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status);
```

- Helps manage progressive completions

```
int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount,  
                int *indices, MPI_Status *statuses);
```

MPI_Test...



- Flag true means completed

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

```
int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses);
```

```
int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag,  
               MPI_Status *status);
```

- Like a non blocking MPI_Waitsome

```
int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indices,  
               MPI_Status *statuses);
```

The Need for MPI_Probe and MPI_Iprobe



- The `MPI_PROBE` and `MPI_IPROBE` operations allow incoming messages to be checked for, without actually receiving them
- The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status)
- In particular, the user may allocate memory for the receive buffer, according to the length of the probed message

Probe to Receive



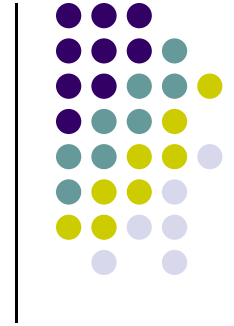
- Probes yield incoming size
- Blocking Probe, wait till match
`int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);`
- Non Blocking Probe, flag true if matching MPI send message in flight
`int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);`

MPI Point-to-Point Communication

~Take Away Slide~



- Two types of communication:
 - Blocking:
 - Safe to change content of buffer holding on to data in the MPI send call
 - Non-blocking:
 - Be careful with the data in the buffer, since you might step on/use it too soon
- MPI provides four modes for these two types
 - standard, synchronous, buffered, ready



Collective Actions



Collective Actions

- MPI actions involving a group of processes
- Must be called by all processes in a communicator
- All collective actions are blocking
- Types of Collective Actions (three of them):
 - Global Synchronization (barrier synchronization)
 - Global Communication (broadcast, scatter, gather, etc.)
 - Global Operations (sum, global maximum, etc.)



Barrier Synchronization

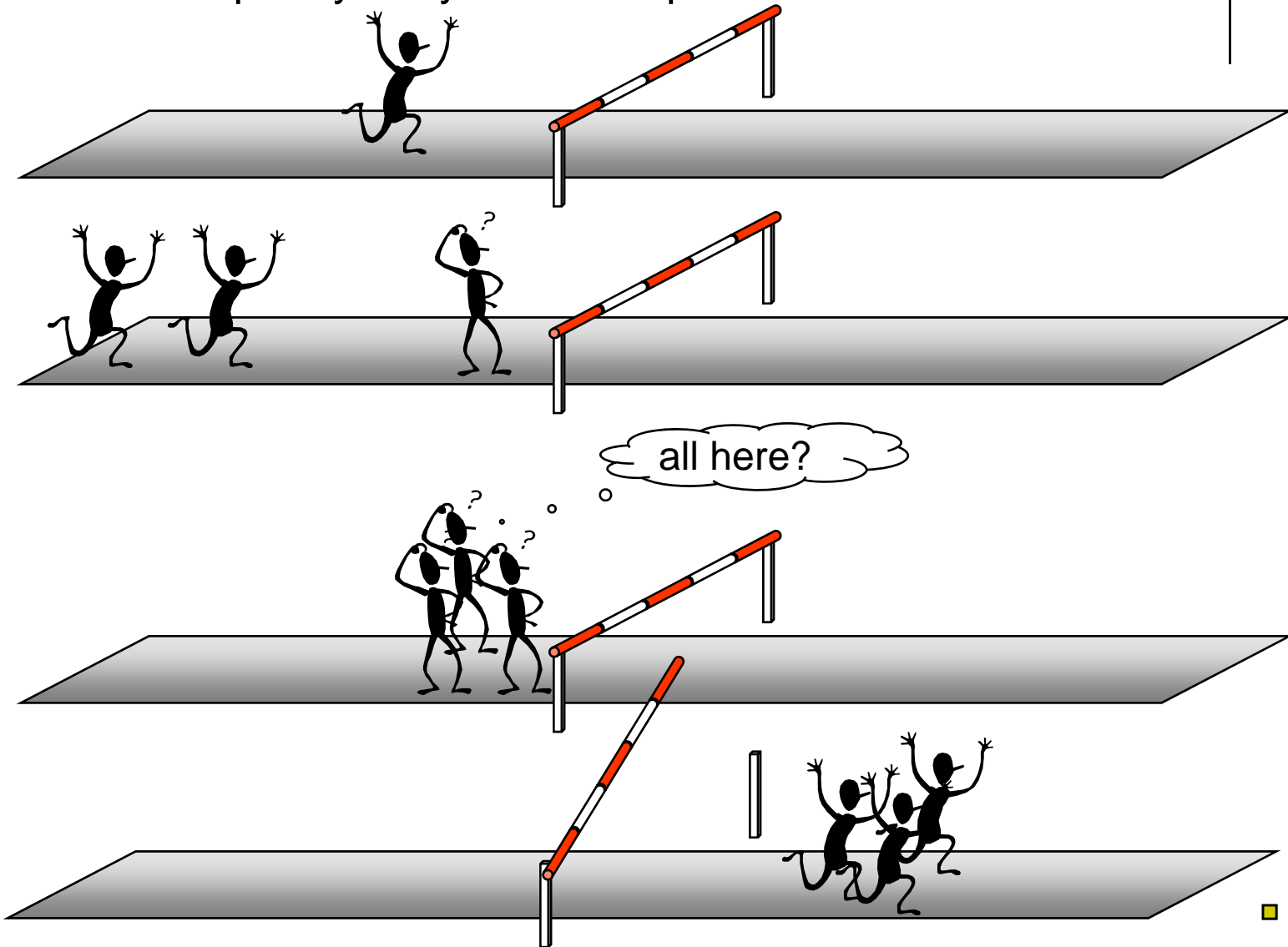
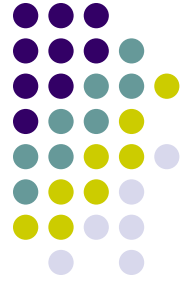
- Syntax:

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

- `MPI_Barrier` not needed that often:
 - Synchronization done automatically for collectives
 - Useful for debugging
 - Remember to remove for production release

Barrier

- Used explicitly to synchronize processes

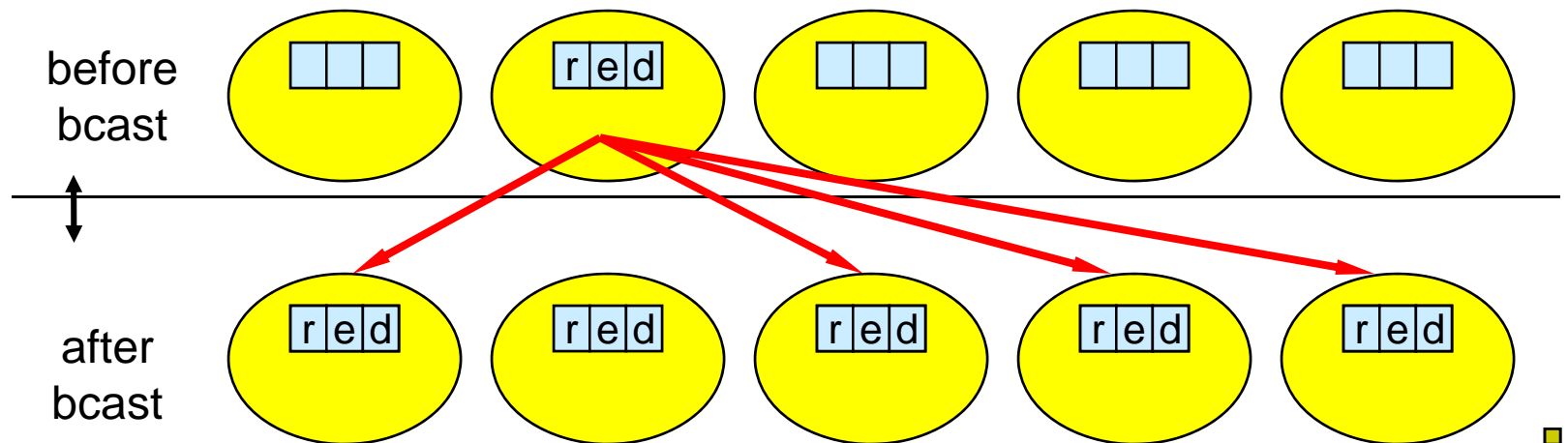


Communication Action: Broadcast



- Function prototype:

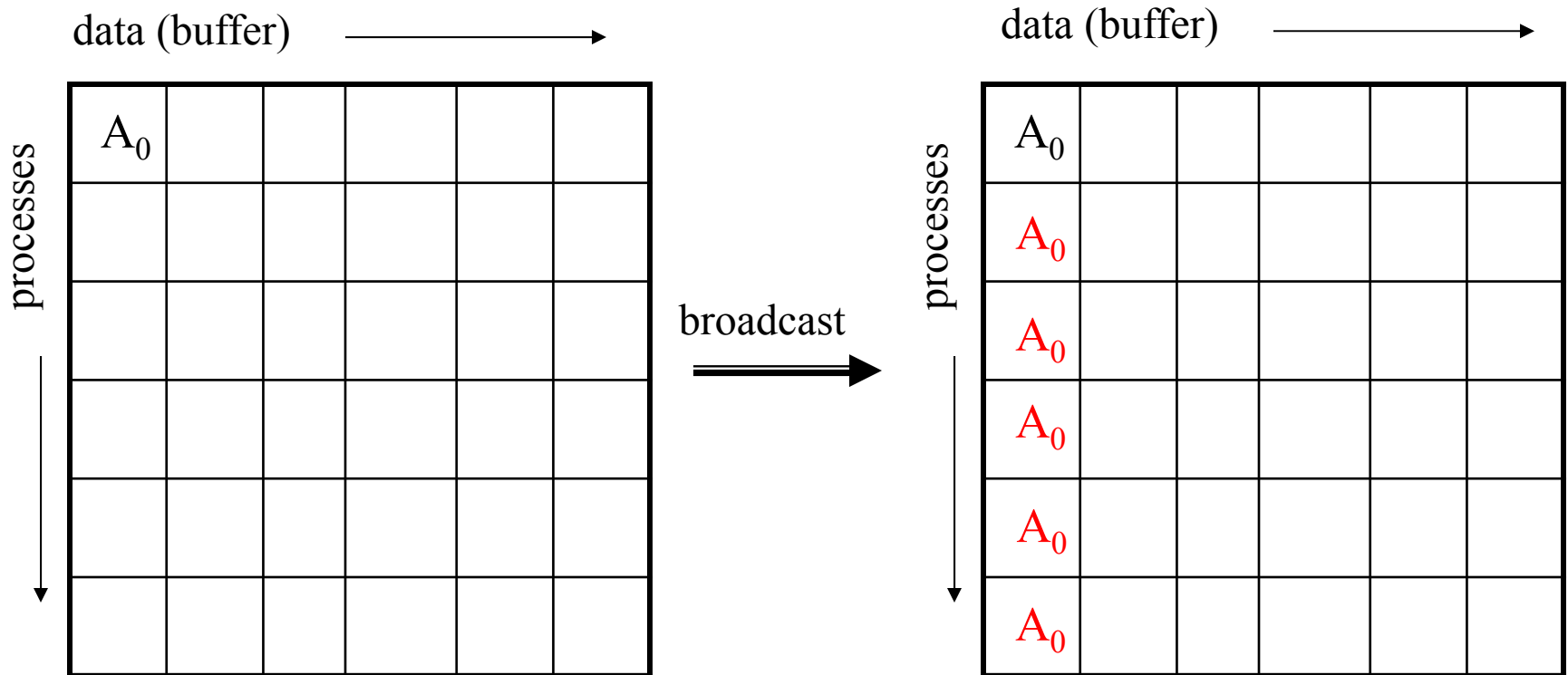
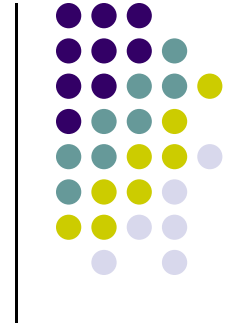
```
int MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```



e.g., root=1

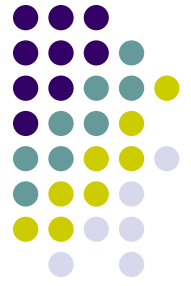
- rank of the sending process (i.e., root process)
- must be given identically by all processes

MPI_Bcast



A_0 : any chunk of contiguous data described with MPI_Datatype and count

MPI_Bcast



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

INOUT : `buffer` (starting address, as usual)
IN : `count` (number of entries in buffer)
IN : `type` (can be user-defined)
IN : `root` (rank of broadcast root)
IN : `com` (communicator)

- Broadcasts message from `root` to all processes (including `root`)
- `com` and `root` must be identical on all processes
- On return, contents of `buffer` is copied to all processes in `com`

Example: MPI_Bcast

- Read a parameter file on a single processor and send data to all processes

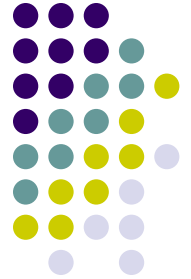
```
#include "mpi.h"
#include <assert.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs;
    float data = -1.0;
    FILE *file;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if( myRank==0 ) {
        char input[100];
        file = fopen("data1.txt", "r");
        assert (file != NULL);
        fscanf(file, "%s\n", input);
        data = atof(input);
    }
    printf("data before: %f\n", data);
    MPI_Bcast(&data, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    printf("data after: %f\n", data);

    MPI_Finalize();
}
```





Example: MPI_Bcast

[Output]

```
[negrut@euler CodeBits]$ qsub -I -l nodes=8:ppn=4,walltime=5:00  
qsub: waiting for job 16114.euler to start  
qsub: job 16114.euler ready
```

```
[negrut@euler17 CodeBits]$ mpicxx testMPI.cpp  
[negrut@euler17 CodeBits]$ mpiexec -np 4 a.out  
data before: -1.000000  
data before: -1.000000  
data before: -1.000000  
data before: 23.330000  
data after: 23.330000  
data after: 23.330000  
data after: 23.330000  
data after: 23.330000
```

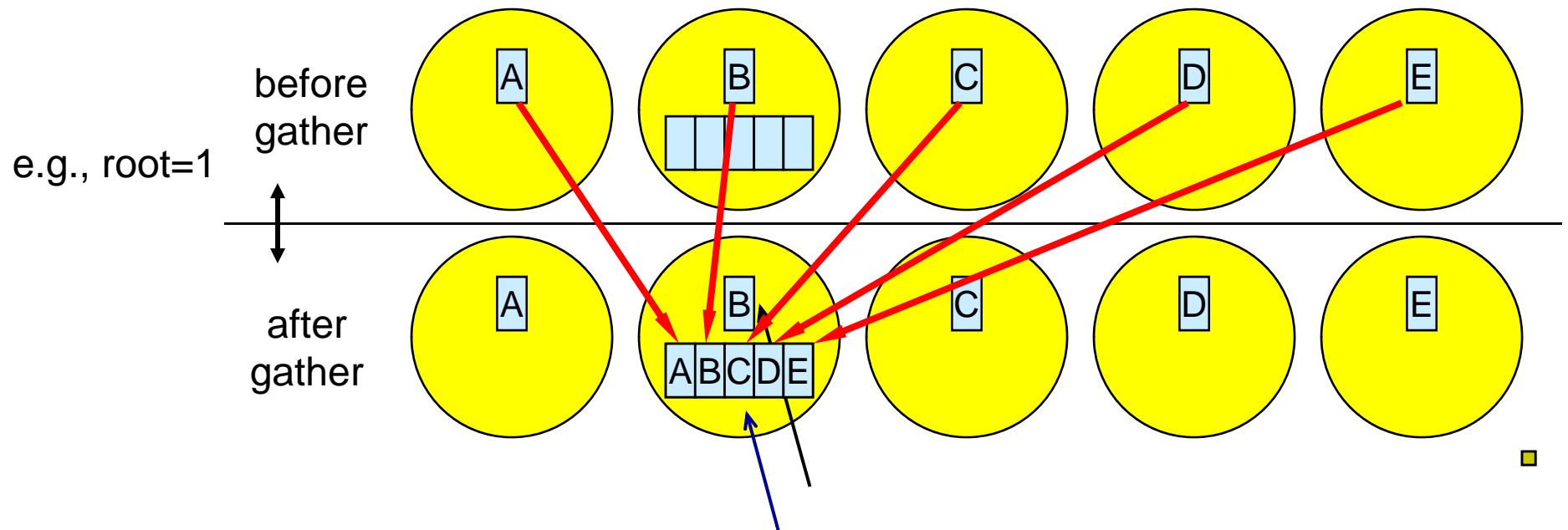


Communication Action: Gather

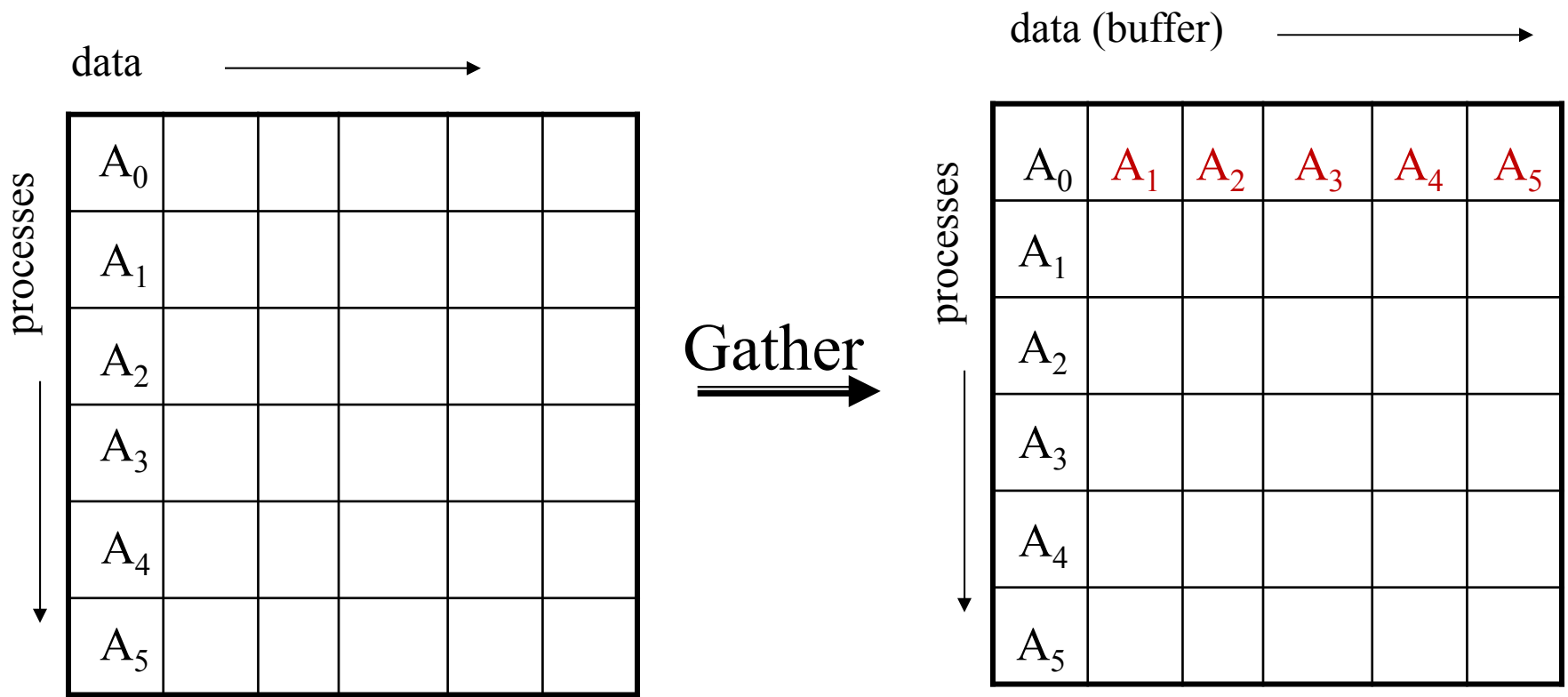
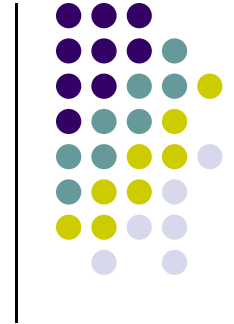


- Function Prototype

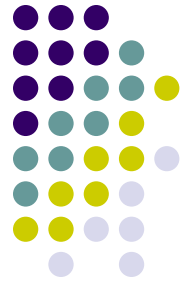
```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,  
              int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



MPI_Gather



MPI_Gather



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

- IN `sendbuf` (starting address of send buffer)
- IN `sendcount` (number of elements in send buffer)
- IN `sendtype` (type)
- OUT `recvbuf` (address of receive buffer)
- IN `recvcount` (n-elements **for any single receive**)
- IN `recvtype` (data type of recv buffer elements)
- IN `root` (rank of receiving process)
- IN `comm` (communicator)

MPI_Gather



- Each process sends content of send buffer to the root process
- Root receives and stores in rank order
- Remarks:
 - Receive buffer argument ignored for all non-root processes (also recvtype, etc.)
 - `recvcount` on root indicates number of items received from each process, not total. This is a common error
- Exercise: Sketch an implementation of `MPI_Gather` using only send and receive operations.



```
#include "mpi.h"
#include <stdlib.h>
int main(int argc, char **argv){
    int myRank, nprocs, nlcl=2, n, i;
    float *data, *data_loc;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    /* local array size on each proc = nlcl */
    data_loc = (float *) malloc(nlcl*sizeof(float));

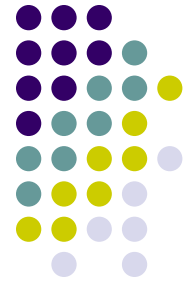
    for (i = 0; i < nlcl; ++i) data_loc[i] = myRank;

    if (myRank == 0) data = (float *) malloc(nprocs*sizeof(float)*nlcl);

    MPI_Gather(data_loc, nlcl, MPI_FLOAT, data, nlcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if (myRank == 0){
        for (i = 0; i < nlcl*nprocs; ++i){
            printf("%f\n", data[i]);
        }
    }

    MPI_Finalize();
    return 0;
}
```



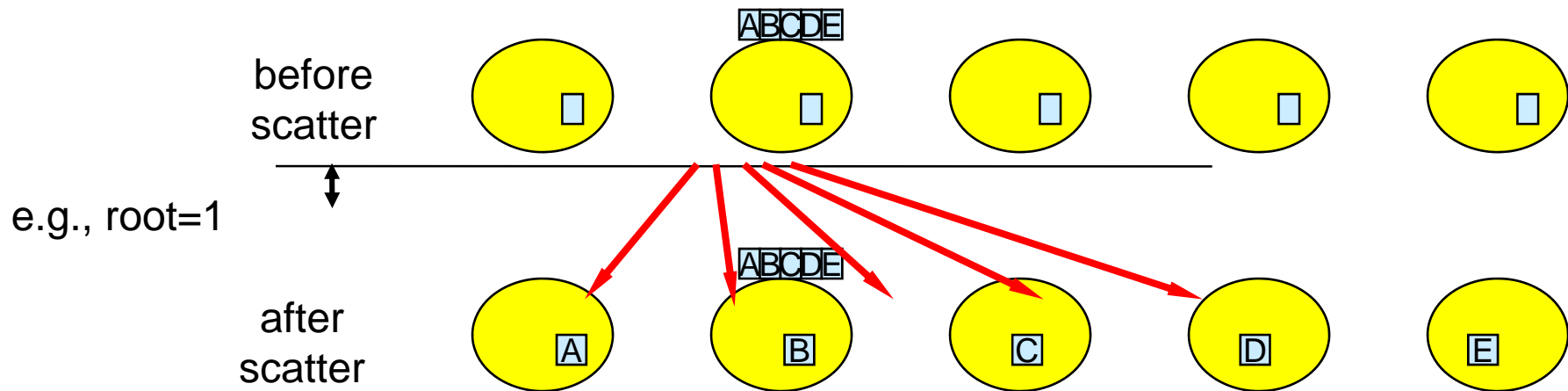
```
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
0.000000
1.000000
1.000000
2.000000
2.000000
3.000000
3.000000
4.000000
4.000000
5.000000
5.000000
[negrut@euler20 CodeBits]$
```

Communication Action: Scatter

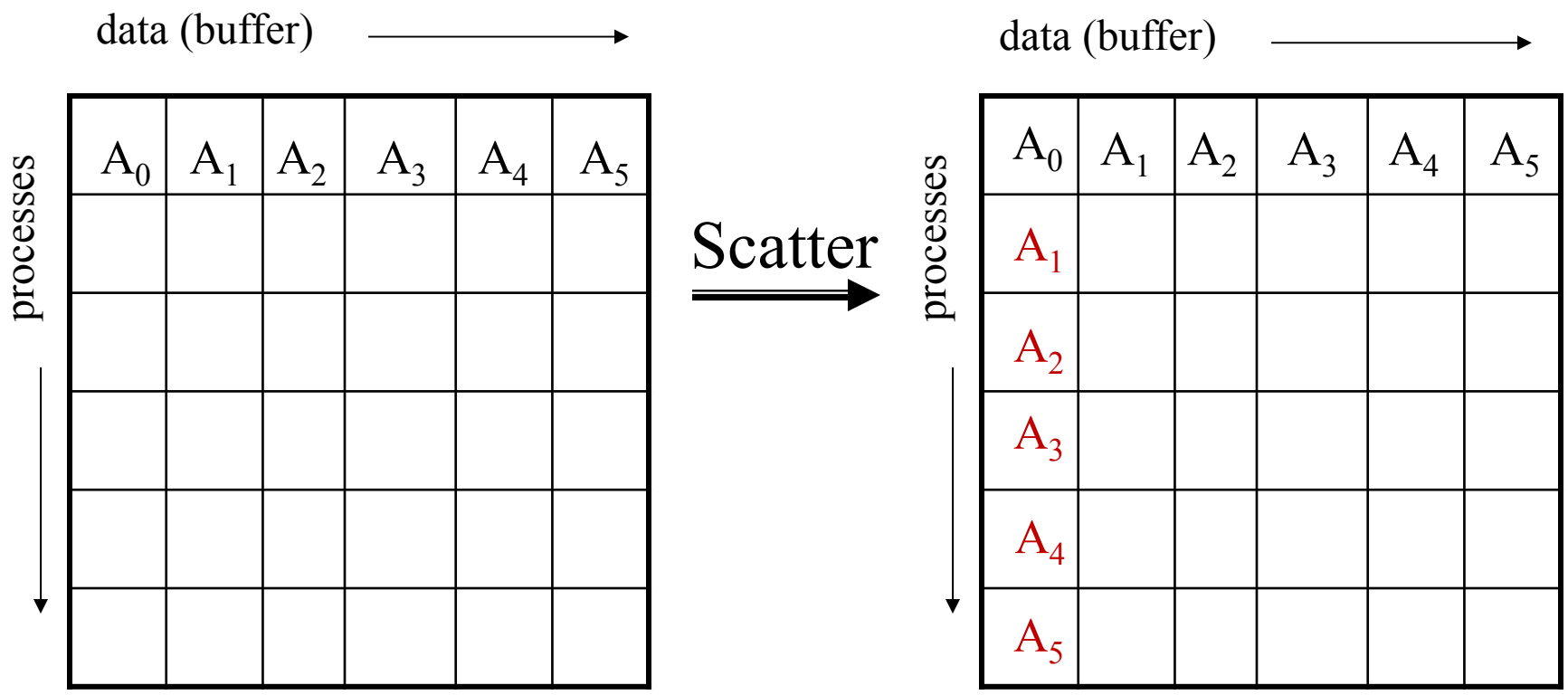
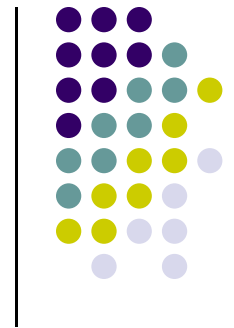


- Function prototype

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



MPI_Scatter



MPI_Scatter



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

- IN `sendbuf` (starting address of send buffer)
- IN `sendcount` (number of elements **sent to each process**)
- IN `sendtype` (type)
- OUT `recvbuf` (address of receive bufer)
- IN `recvcount` (n-elements **in receive buffer**)
- IN `recvtype` (data type of receive elements)
- IN `root` (rank of sending process)
- IN `comm` (communicator)

MPI_Scatter



- Inverse of **MPI_Gather**
- Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter
- Remarks:
 - All arguments are significant on **root**, while on other processes only **recvbuf**, **recvcount**, **recvtype**, **root**, and **comm** are significant

```

#include "mpi.h"
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, n_lcl=2;
    float *data, *data_l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    /* local array size on each proc = n_lcl */
    data_l = (float *) malloc(n_lcl*sizeof(float));

    if( myRank==0 ) {
        data = (float *) malloc(nprocs*sizeof(float)*n_lcl);
        for( int i = 0; i < nprocs*n_lcl; ++i) data[i] = i;
    }

    MPI_Scatter(data, n_lcl, MPI_FLOAT, data_l, n_lcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

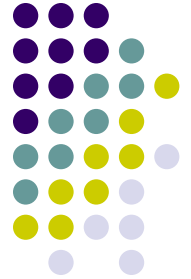
    for( int n=0; n < nprocs; ++n ){
        if( myRank==n ){
            for (int j = 0; j < n_lcl; ++j) printf("%f\n", data_l[j]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}

```

This is interesting.
Think what's happening
here...





```
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
10.000000
11.000000
[negrut@euler20 CodeBits]$
```

Putting Things in Perspective...



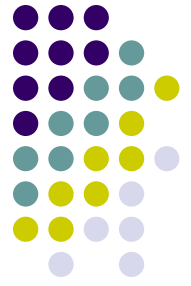
- Gather: you automatically create a serial array from a distributed one
- Scatter: you automatically create a distributed array from a serial one



Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- $d_0 \circ d_1 \circ d_2 \circ d_3 \circ \dots \circ d_{s-2} \circ d_{s-1}$
 - d_i = data in process rank i
 - single variable, or
 - vector
 - \circ = associative operation
 - Example:
 - global sum or product
 - global maximum or minimum
 - global user-defined operation
- Floating point rounding may depend on usage of associative law:
 - $[(d_0 \circ d_1) \circ (d_2 \circ d_3)] \circ [\dots \circ (d_{s-2} \circ d_{s-1})]$
 - $(((((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \dots) \circ d_{s-2}) \circ d_{s-1}))$

Example of Global Reduction

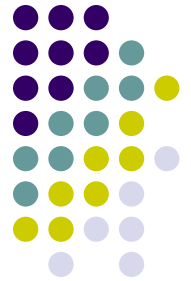


- Global integer sum
- Sum of all `inbuf` values should be returned in `resultbuf`.
- Assume `root=0`;

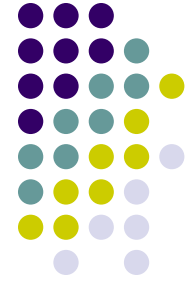
```
MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```

- The result is only placed in `resultbuf` at the root process.

Predefined Reduction Operation Handles

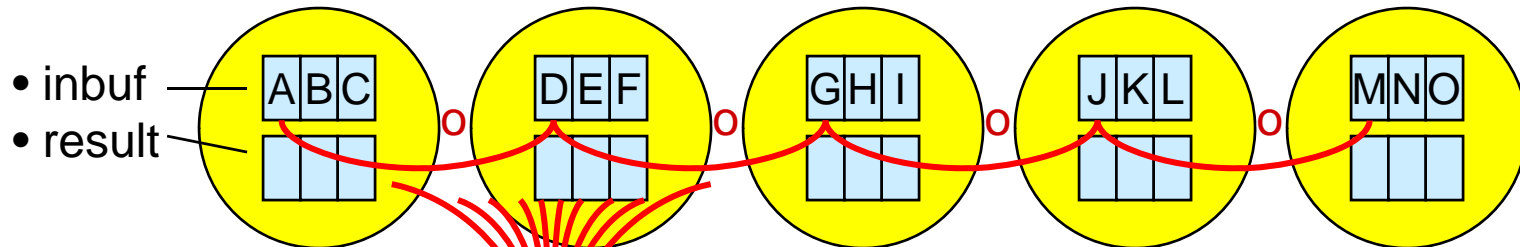


Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum ⁵⁵

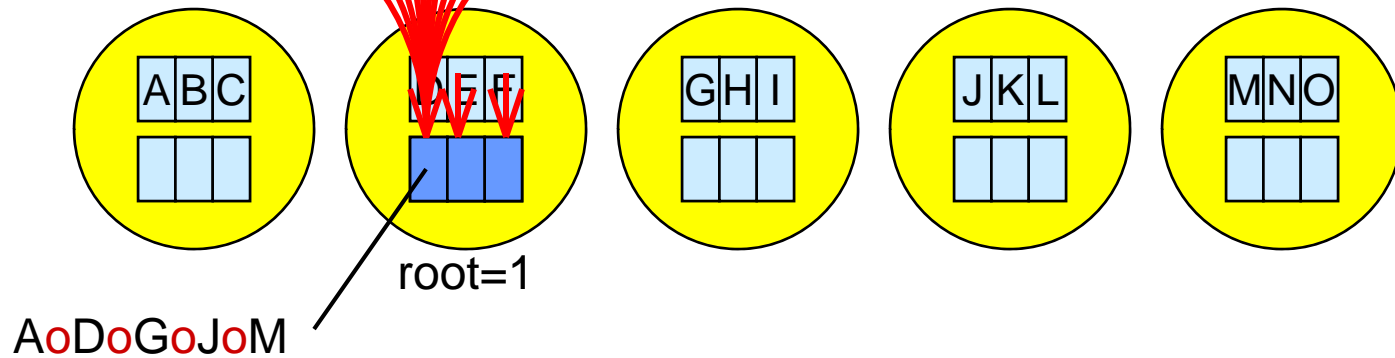


MPI_Reduce

before MPI_REDUCE

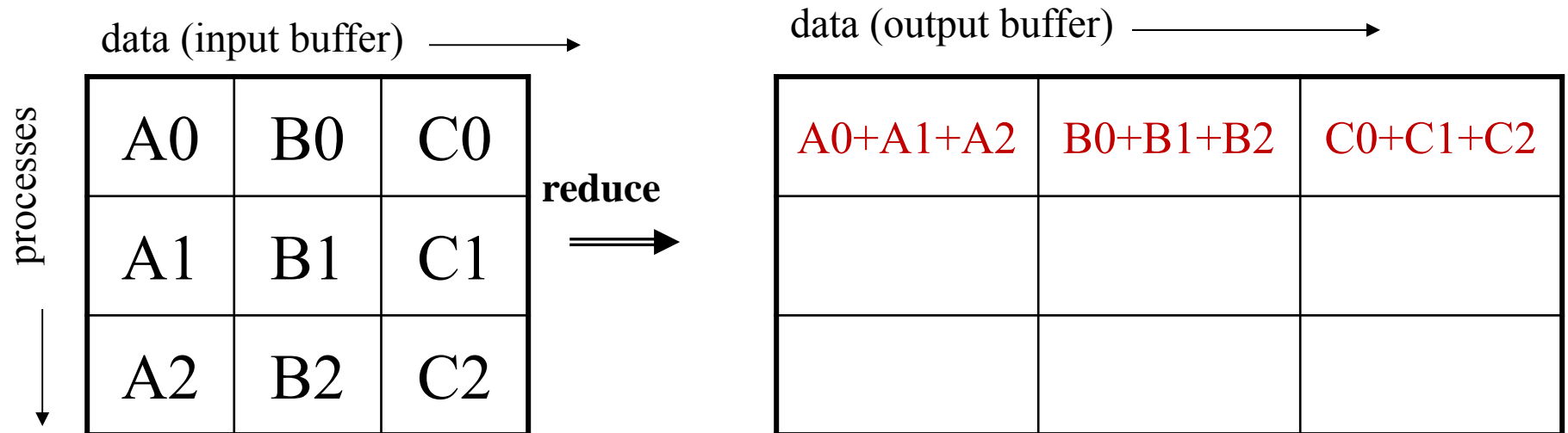


after





Reduce Operation



Assumption: Rank 0 is the root

MPI_Reduce



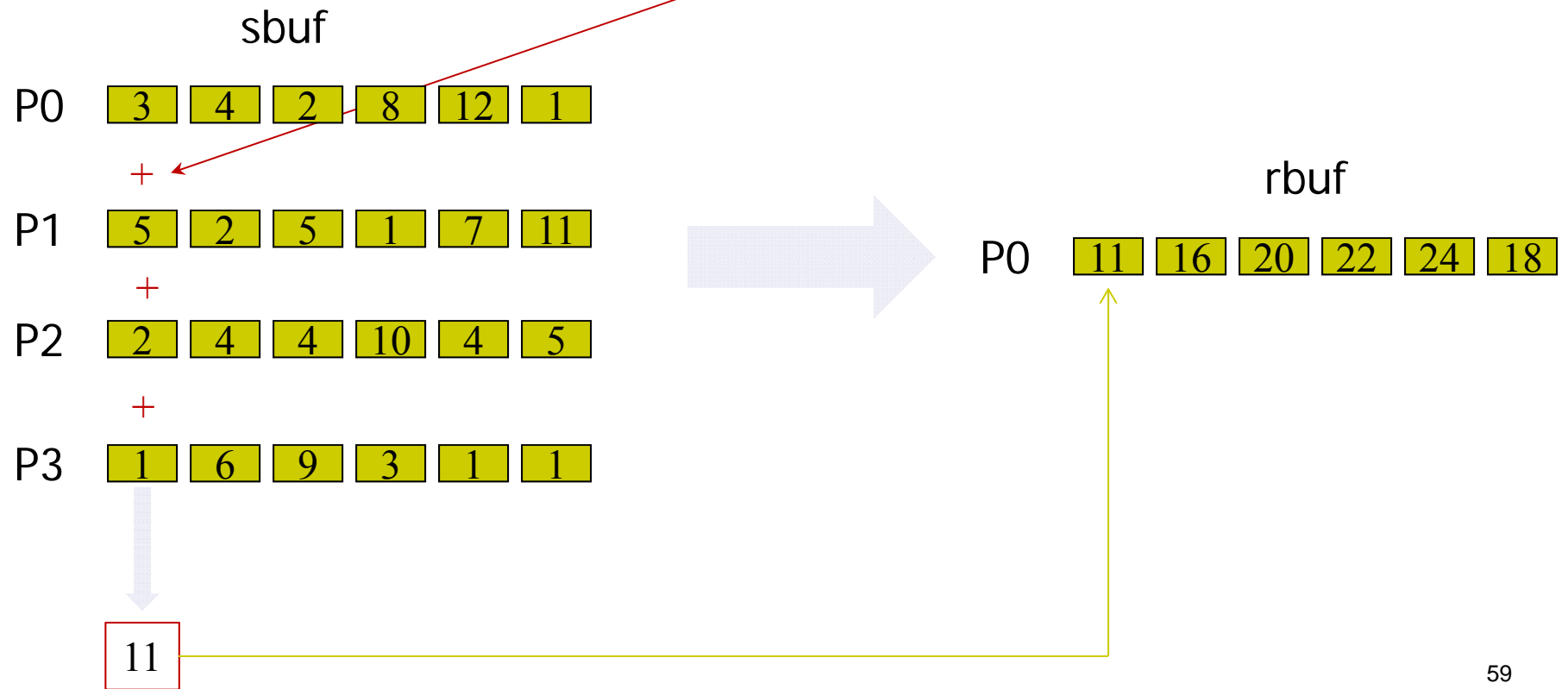
```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count,  
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

- IN `sendbuf` (address of send buffer)
- OUT `recvbuf` (address of receive buffer)
- IN `count` (number of elements in send buffer)
- IN `datatype` (data type of elements in send buffer)
- IN `op` (reduce operation)
- IN `root` (rank of root process)
- IN `comm` (communicator)



MPI_Reduce example

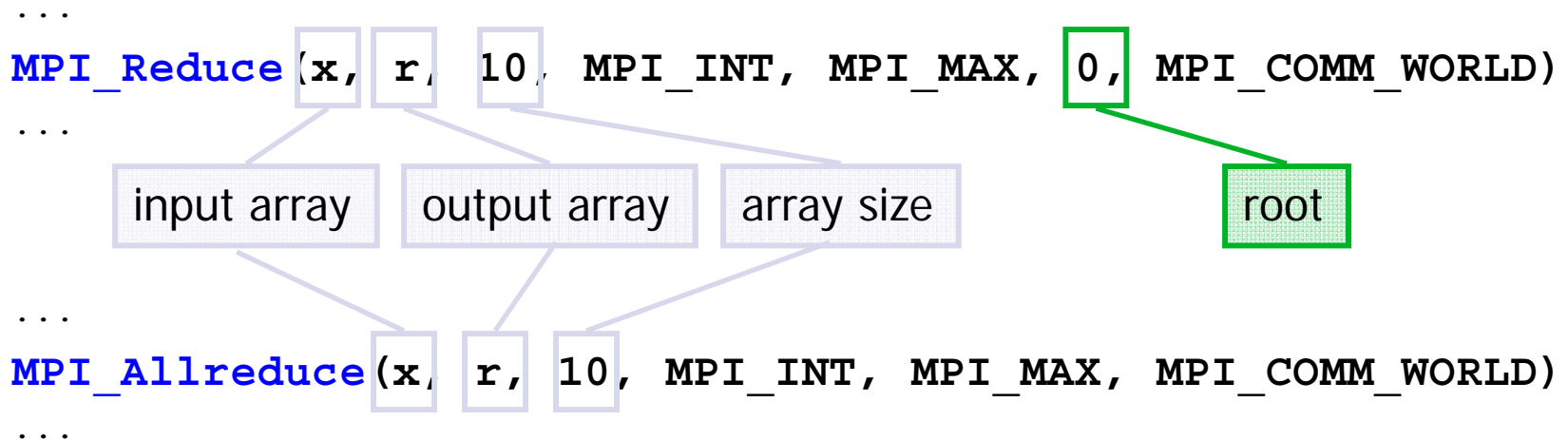
```
MPI_Reduce(sbuf, rbuf, 6, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```



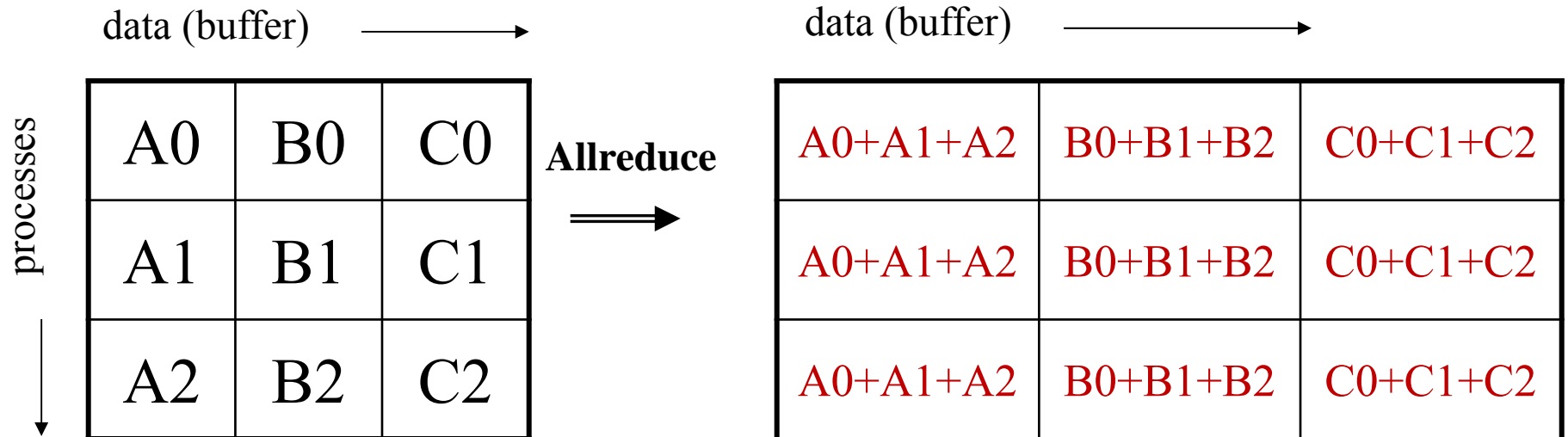
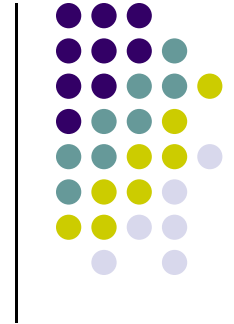


MPI_Reduce, MPI_Allreduce

- **MPI_Reduce**: result is collected by the root only
 - The operation is applied element-wise for each element of the input arrays on each processor
- **MPI_Allreduce**: result is sent out to everyone



MPI_Allreduce





MPI_Allreduce

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

- IN `sendbuf` (address of send buffer)
- OUT `recvbuf` (address of receive buffer)
- IN `count` (number of elements in send buffer)
- IN `datatype` (data type of elements in send buffer)
- IN `op` (reduce operation)
- IN `comm` (communicator)



Example: MPI_Allreduce

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int my_rank, nprocs, gsum, gmax, gmin, data_1;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    data_1 = my_rank;

    MPI_Allreduce(&data_1, &gsum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&data_1, &gmax, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    MPI_Allreduce(&data_1, &gmin, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

    printf("gsum: %d, gmax: %d gmin:%d\n", gsum,gmax,gmin);
    MPI_Finalize();
}
```



Example: MPI_Allreduce

[Output]

```
[negrut@euler24 CodeBits]$ mpiexec -np 10 me759.exe
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
[negrut@euler24 CodeBits]$
```


MPI_SCAN



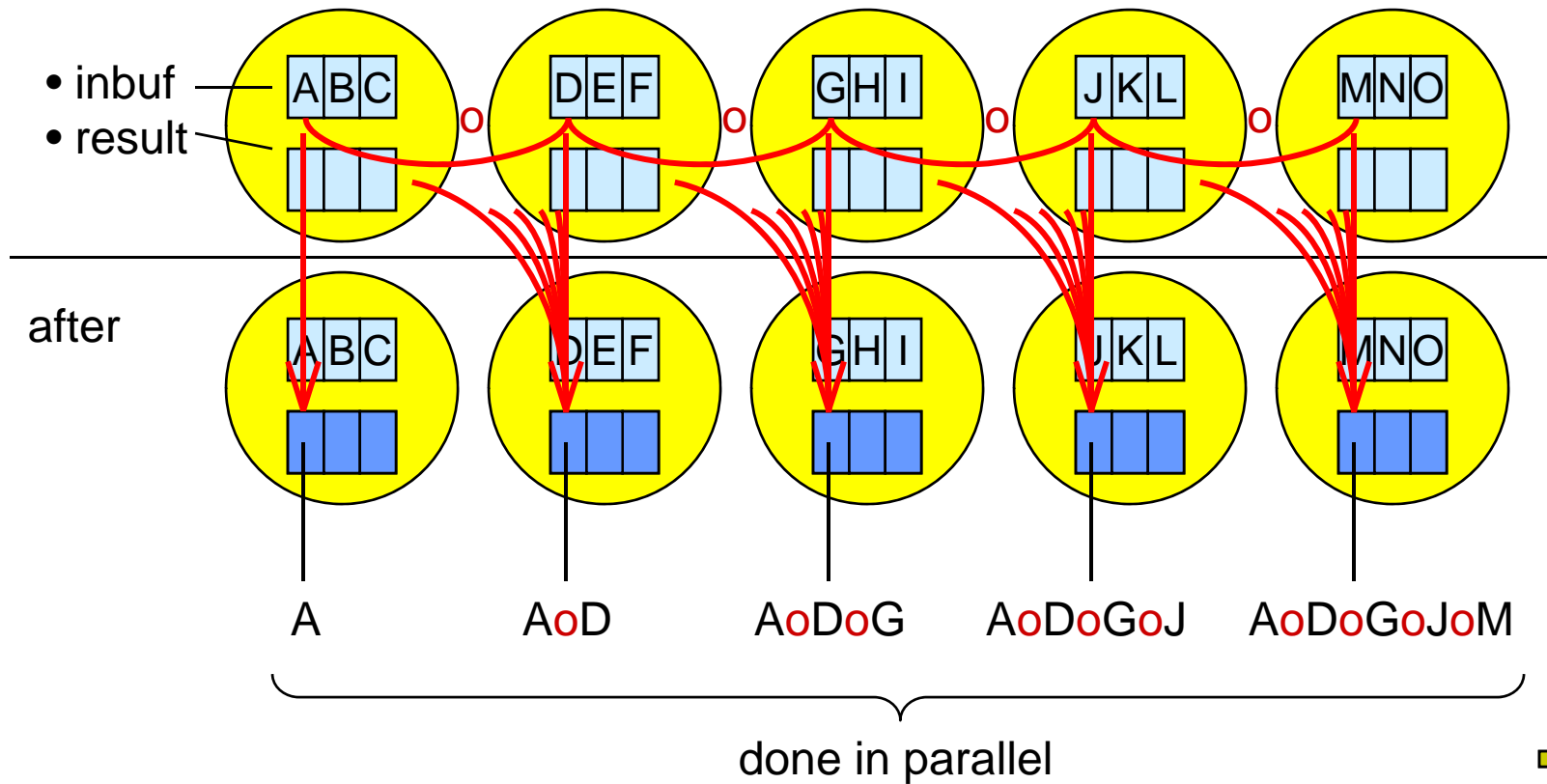
- Performs a prefix reduction on data distributed across a communicator
- The operation returns, in the receive buffer of the process with rank i , the reduction of the values in the send buffers of processes with ranks $0, \dots, i$ (inclusive)
- The type of operations supported, their semantics, and the constraints on send and receive buffers are as for **MPI_REDUCE**

MPI_SCAN

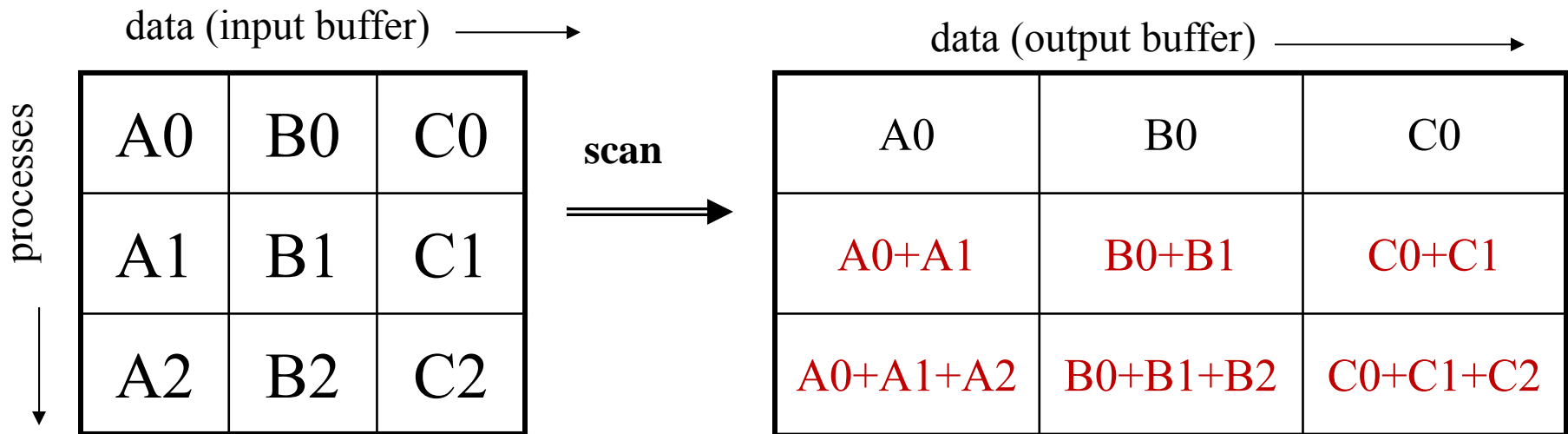
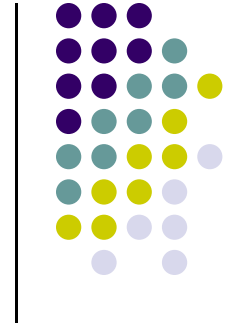


before MPI_SCAN

- inbuf
- result



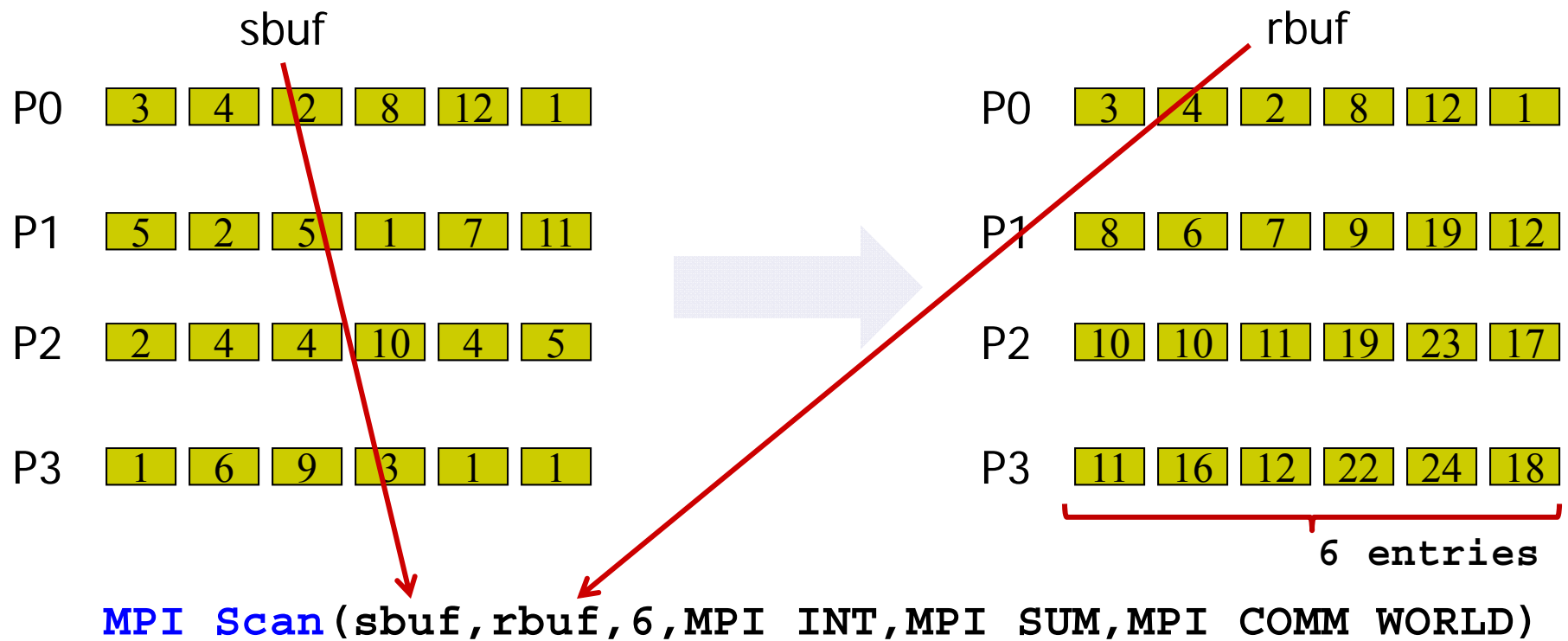
Scan Operation





MPI_Scan: Prefix reduction

- Process i receives data reduced on process 0 through i



MPI_Scan



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

- IN `sendbuf` (address of send buffer)
 - OUT `recvbuf` (address of receive buffer)
 - IN `count` (number of elements in send buffer)
 - IN `datatype` (data type of elements in send buffer)
 - IN `op` (reduce operation)
 - IN `comm` (communicator)
-
- Note: `count` refers to total number of elements that will be received into receive buffer after operation is complete

```

#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, i, n;
    int *result, *data_l;
    const int dimArray = 2;

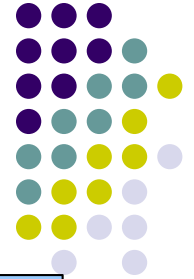
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    data_l = (int *) malloc(dimArray*sizeof(int));
    for (i = 0; i < dimArray; ++i) data_l[i] = (i+1)*myRank;
    for (n = 0; n < nprocs; ++n) {
        if( myRank == n ) {
            for(i=0; i<dimArray; ++i) printf("Process %d. Entry: %d. Value: %d\n", myRank, i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    result = (int *) malloc(dimArray*sizeof(int));
    MPI_Scan(data_l, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (n = 0; n < nprocs; ++n){
        if (myRank == n) {
            printf("\n Post Scan - Content on Process: %d\n", myRank);
            for (i = 0; i < dimArray; ++i) printf("Entry: %d. Scan Val: %d\n", i, result[i]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    free(result); free(data_l);
    return 0;
}

```



Example: MPI_Scan

[Output]

```
[negrut@euler26 CodeBits]$ mpicxx -o me759.exe testMPI.cpp
```

```
[negrut@euler26 CodeBits]$ mpiexec -np 4 me759.exe
```

```
Process 0. Entry: 0. Value: 0
```

```
Process 0. Entry: 1. Value: 0
```

```
Process 1. Entry: 0. Value: 1
```

```
Process 1. Entry: 1. Value: 2
```

```
Process 2. Entry: 0. Value: 2
```

```
Process 2. Entry: 1. Value: 4
```

```
Process 3. Entry: 0. Value: 3
```

```
Process 3. Entry: 1. Value: 6
```

```
Post Scan - Content on Process: 0
```

```
Entry: 0. Scan Val: 0
```

```
Entry: 1. Scan Val: 0
```

```
Post Scan - Content on Process: 1
```

```
Entry: 0. Scan Val: 1
```

```
Entry: 1. Scan Val: 2
```

```
Post Scan - Content on Process: 2
```

```
Entry: 0. Scan Val: 3
```

```
Entry: 1. Scan Val: 6
```

```
Post Scan - Content on Process: 3
```

```
Entry: 0. Scan Val: 6
```

```
Entry: 1. Scan Val: 12
```

```
[negrut@euler26 CodeBits]$
```

MPI_Exscan



- **MPI_Exscan** is like **MPI_Scan**, except that the contribution from the calling process is not included in the result at the calling process (it is contributed to the subsequent processes)
- The value in **recvbuf** on the process with rank 0 is undefined, and **recvbuf** is not significant on process 0
- The value in **recvbuf** on the process with rank 1 is defined as the value in **sendbuf** on the process with rank 0
- For processes with rank $i > 1$, the operation returns, in the receive buffer of the process with rank i , the reduction of the values in the send buffers of processes with ranks $0, \dots, i-1$ (inclusive)
- The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for **MPI_REDUCE**

MPI_Exscan



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

- IN `sendbuf` (address of send buffer)
- OUT `recvbuf` (address of receive buffer)
- IN `count` (number of elements in send buffer)
- IN `datatype` (data type of elements in send buffer)
- IN `op` (reduce operation)
- IN `comm` (communicator)

```

#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs,i, n;
    int *result, *data_l;
    const int dimArray = 2;

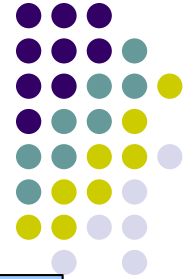
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    data_l = (int *) malloc(dimArray*sizeof(int));
    for (i = 0; i < dimArray; ++i) data_l[i] = (i+1)*myRank;
    for (n = 0; n < nprocs; ++n){
        if( myRank == n ) {
            for(i=0; i<dimArray; ++i) printf("Process %d. Entry: %d. Value: %d\n", myRank, i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    result = (int *) malloc(dimArray*sizeof(int));
    MPI_Exscan(data_l, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (n = 0; n < nprocs; ++n){
        if (myRank == n) {
            printf("\n Post Scan - Content on Process: %d\n", myRank);
            for (i = 0; i < dimArray; ++i) printf("Entry: %d. Scan Val: %d\n", i, result[i]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}

```



Example: MPI_Exscan

[Output]

```
[negrut@euler26 CodeBits]$ mpicxx -o me759.exe testMPI.cpp
```

```
[negrut@euler26 CodeBits]$ mpiexec -np 4 me759.exe
```

```
Process 0. Entry: 0. Value: 0
```

```
Process 0. Entry: 1. Value: 0
```

```
Process 1. Entry: 0. Value: 1
```

```
Process 1. Entry: 1. Value: 2
```

```
Process 2. Entry: 0. Value: 2
```

```
Process 2. Entry: 1. Value: 4
```

```
Process 3. Entry: 0. Value: 3
```

```
Process 3. Entry: 1. Value: 6
```

```
Post Scan - Content on Process: 0
```

```
Entry: 0. Scan Val: 321045752
```

```
Entry: 1. Scan Val: 32593
```

```
Post Scan - Content on Process: 1
```

```
Entry: 0. Scan Val: 0
```

```
Entry: 1. Scan Val: 0
```

```
Post Scan - Content on Process: 2
```

```
Entry: 0. Scan Val: 1
```

```
Entry: 1. Scan Val: 2
```

```
Post Scan - Content on Process: 3
```

```
Entry: 0. Scan Val: 3
```

```
Entry: 1. Scan Val: 6
```

```
[negrut@euler26 CodeBits]$
```

User-Defined Reduction Operations



- Operator handles
 - Predefined – see table of last lecture: MPI_SUM, MPI_MAX, etc.
 - User-defined
- User-defined operation ■:
 - Should be associative
 - User-defined function must perform the operation “vector_A ■ vector_B”

- Registering a user-defined reduction function:

```
MPI_Op_create( MPI_User_function *func, int commute, MPI_Op *op);
```

- `commute` tells the MPI library whether `func` is commutative or not

Example: Norm 1 of a Vector

```
int main(int argc, char* argv[]) {  
    int root=0, p, myid;  
    float sendbuf, recvbuf;  
    MPI_Op myop;
```

```
    int commutes=1;  
    MPI_Init(&argc, &argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &p);  
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
    //create the operator...  
    MPI_Op_create(onenorm, commutes, &myop);
```

```
    //get some fake data used to make the point...  
    sendbuf = myid*(-1)^myid;  
    MPI_Barrier(MPI_COMM_WORLD);
```

```
    MPI_Reduce (&sendbuf, &recvbuf, 1, MPI_FLOAT, myop, root, MPI_COMM_WORLD);  
    if( myid == root )  
        printf("The operation yields %f\n", recvbuf);  
    MPI_Finalize();  
    return 0;
```

```
}
```

```
#include <mpi.h>  
#include <stdio>  
#include <math.h>  
  
void oneNorm(float *in, float *inout, int *len,  
             MPI_Datatype *type)  
{  
    int i;  
    for (i=0; i<*len; i++) {  
        *inout = fabs(*in) + fabs(*inout);    /* one-norm */  
        in++;  
        inout++;  
    }  
}
```

 Continues here...

thrust code more simple...



```
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>

template <typename T> struct absval {
    __host__ __device__
    T operator()(const T& x) const {
        return fabs(x);
    }
};

int main(void)
{
    // initialize host array
    float x[4] = {1.0, -2.0, 3.0, -4.0};

    // transfer to device
    thrust::device_vector<float> d_x(x, x + 4);

    absval<float> unary_op;
    float res = thrust::transform_reduce(d_x.begin(), d_x.end(), unary_op, 0.f, thrust::plus<float>());

    std::cout << res << std::endl;
    return 0;
}
```