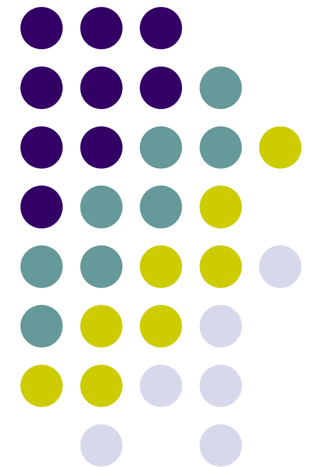
ME759
High Performance Computing for Engineering Applications

Parallel Computing with the Message Passing Interface (MPI)

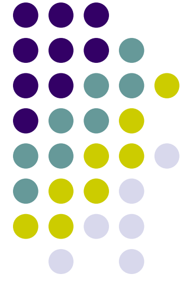
November 4, 2013



© Dan Negrut, 2013
ME964 UW-Madison

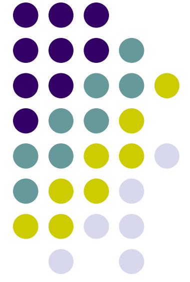
"You know that uncertainty you feel today? It never goes away. The question is, do you know how to make uncertainty your friend? "
– David Brooks, journalist, The New York Times

Before We Get Started…



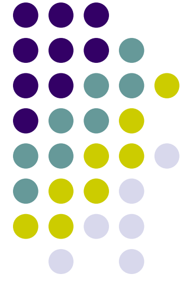
- Last time:
  - MPI practicalities: compiling and running MPI application on Euler
  - Point-to-point communication in MPI: blocking flavors of send/receive
- Today:
  - Wrap up point-to-point communication in MPI: non-blocking flavors
  - Collective action: barriers, communication, operations
- Miscellaneous
  - HW due tonight at 11:59 PM. Most challenging assignment of ME759
  - New assignment posted later today. Due in one week
    - Has to do with thrust
  - Last regular lecture is on Wd. Fr lecture set aside for Midterm Exam

Midterm & Final Project Partitioning



- If you are happy with your Midterm Project, it can become your Final Project
  - A midterm project report will be due nonetheless to show adequate progress
  - Intermediate report in this case should be a formality
- If not happy w/ your Midterm Project selection, Nov. 15 provides the opportunity to bail out
  - Report should be detailed and follow rules spelled out in forum posting
- For SPH default project: the student[s] w/ the fastest implementation will write a paper with Arman, Dan and another lab member
- Please post related questions on forum
- See syllabus for deadlines

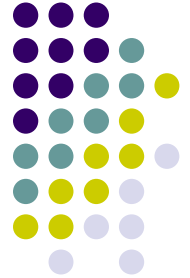
Blocking Type: Communication Modes



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

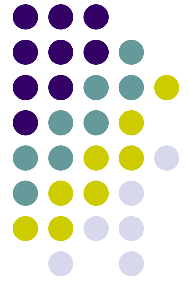
- Receiving all modes → `MPI_RECV`

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	

1) Synchronous Sending in MPI
2) Buffered Sending in MPI

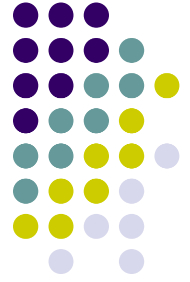


- Synchronous with MPI_Ssend
  - In synchronous mode, a send will not complete until a matching receive is posted.
    - The sender has to wait for a receive to be posted
    - No buffering of data
    - Used for ensuring the code is healthy and doesn’t rely on buffering

- 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 bsends and no matching receives have been posted yet

[A. Snavely]→

3) Standard Sending in MPI
4) Ready Sending in MPI

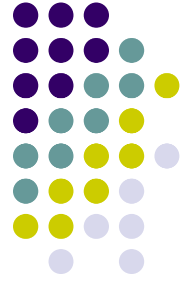


- Standard with MPI_Send
  - Up to the MPI implementation to decide whether to do rendesvous or eager, for performance reasons
    - NOTE: If it does rendezvous, in fact the behavior is that of MPI_SSend
  - Very commonly used

- 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

[A. Snavely]→

Most Important Issue: Deadlocking

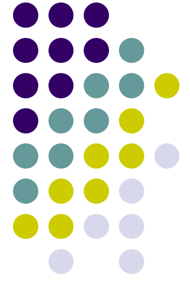


- Deadlock situations: appear when due to a certain sequence of commands the execution hangs

PROCESS 0		PROCESS 1
... MPI_Ssend() MPI_Recv() ...	Deadlock	... MPI_Ssend() MPI_Recv() ...
... MPI_Buffer_attach() MPI_Bsend() MPI_Recv() ...	No Deadlock	... MPI_Buffer_attach() MPI_Bsend() MPI_Recv() ...
... MPI_Buffer_attach() MPI_Bsend() MPI_Recv() ...	No Deadlock	... MPI_Ssend() MPI_Recv() ...

[A. Snavely]→

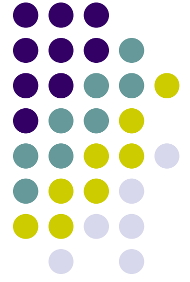
Deadlocking, Another Example



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

[A. Snavely]→

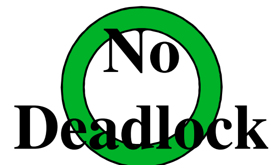
Avoiding Deadlocking



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

PROCESS 0

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



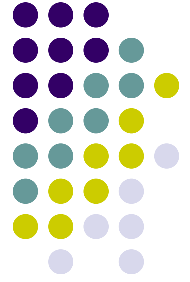
PROCESS 1

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

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

[A. Snavely]→

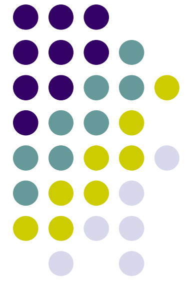
Example



- Always succeeds, even if no buffering is done

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

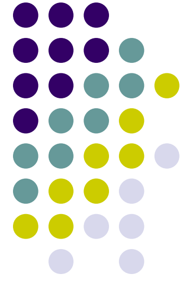
Example



- Will always deadlock, no matter the buffering mode

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

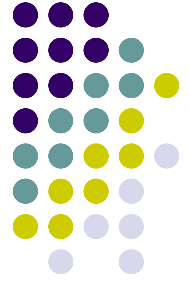
Example



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

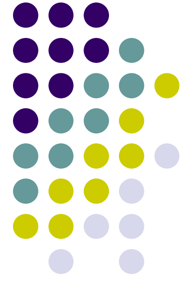
```
if(rank==0)
{
        MPI_Send(...);
        MPI_Recv(...);
}
else if(rank==1)
{
        MPI_Send(...);
        MPI_Recv(...);
}
```

Concluding Remarks, Blocking Options



- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode
  - → risks with synchronous send
- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting → idle time
  - high latency / best bandwidth
- 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

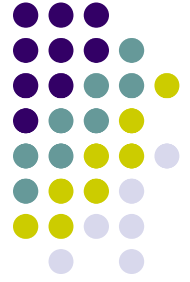


- Relies on the existence of a buffer, which is set up through a call

```
int MPI_Buffer_attach(void* buffer, int size);
```

- A bsend is a <u>local</u> 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 may remain in the buffer until a matching receive is posted

Technicalities, Loose Ends: More on the Buffered Send [Cntd.]



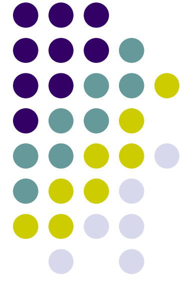
- Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is there is not enough buffer space

- The 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 (MPI_BSEND_OVERHEAD*number_of_messages) should be sufficient

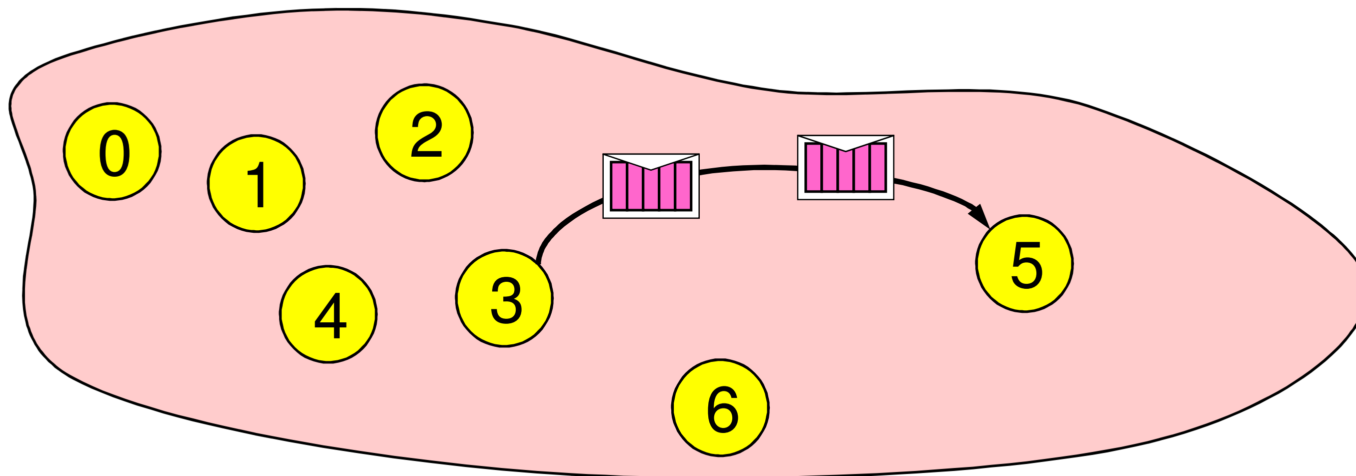
- `MPI_Bsend` lowers bandwidth since it requires an extra memory-to-memory copy of the outgoing data

- The `MPI_Buffer_attach` subroutine provides MPI a buffer in the user's memory. This buffer is used only by messages sent in buffered mode, and only one buffer is attached to a process at any time

Technicalities, Loose Ends: Message Order Preservation



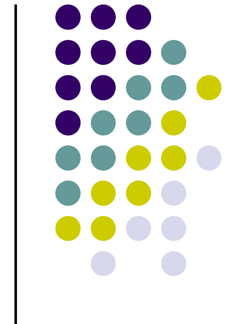
- Rule for messages on the same connection; i.e., same communicator, source, and destination rank:
  - **Messages do not overtake each other**
  - True even for non-synchronous sends

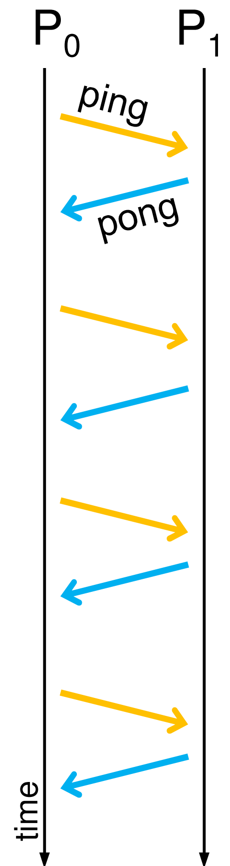


- If both receives match both messages, then the order is preserved

[ICHEC]→

Read This for Assignment 11





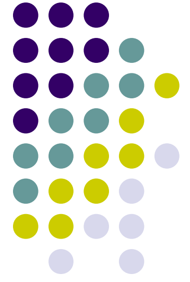
- Write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message, process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Add timing calls before and after the loop
- For timing purposes, 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

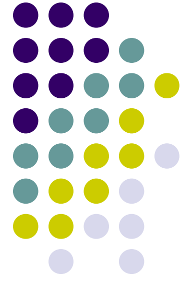
[Useful, for Assignment 11]



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

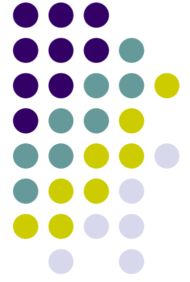
- Resolution is typically 1E-3 seconds
- Time of different processes might actually be synchronized, controlled by the variable `MPI_WTIME_IS_GLOBAL`

More on Timing
[Useful, for Assignment 11; Cntd.]



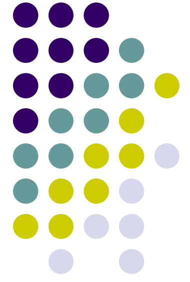
- Latency = transfer time for zero length messages
- Bandwidth = message size (in bytes) / transfer time

- 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.



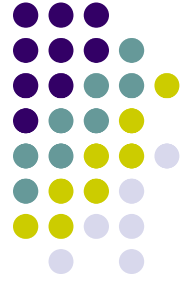
Non-Blocking Communication

Non-Blocking Communications: Motivation



- Overlap communication with execution (just like w/ CUDA):
  - Initiate non-blocking communication
    - Returns **I**mmediately
    - Routine name starting with MPI_**I**…
  - Do some work
    - "latency hiding"
  - Wait for non-blocking communication to complete

Non-blocking Send/Receive



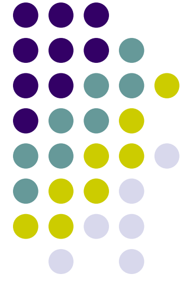
- Syntax

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

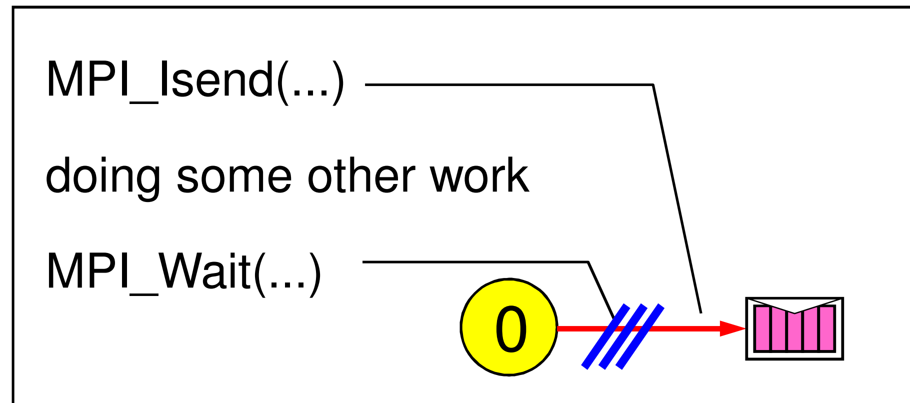
- buf - [in] initial address of send buffer (choice)
- count - [in] number of elements in send buffer (integer)
- datatype - [in] datatype of each send buffer element (handle)
- dest - [in] rank of destination (integer)
- tag - [in] message tag (integer)
- comm - [in] communicator (handle)
- request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

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

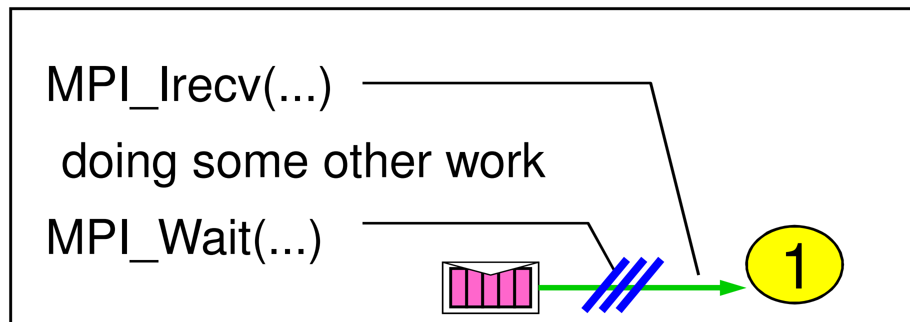
The Screenplay: Non-Blocking P2P Communication



- Non-blocking send



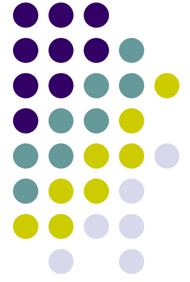
- Non-blocking receive



[ICHEC]→

Non-Blocking Send/Receive

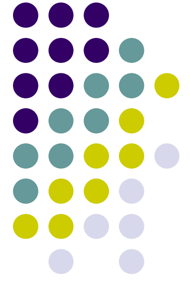
Some Tools of the Trade



- Call returns immediately. Therefore, user must worry whether …
  - Data to be sent is out of the send buffer before trampling on the buffer
  - 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

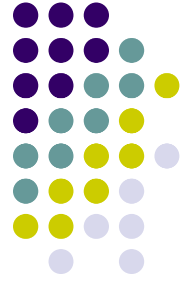
```
int MPI_Waitany(int count, MPI_Request *requests,int *index, MPI_Status *status);
```

  - Helps manage progressive completions

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


[Alexander]→

MPI_Test...



- Flag true means completed

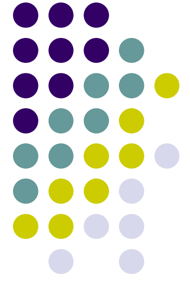
```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);

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);
```

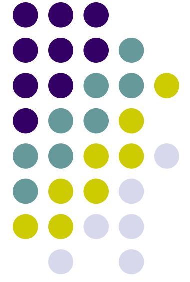

[Alexander]→

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

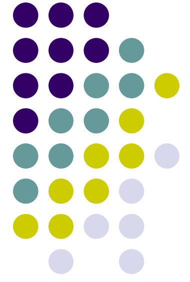
```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- Non Blocking Probe, flag true if ready

```
int MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag,MPI_Status *status);
```

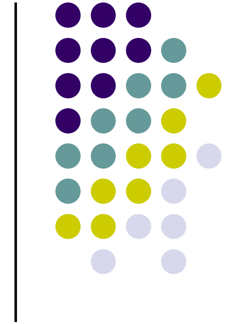

[Alexander]→

MPI Point-to-Point Communication ~Take Away Slide~



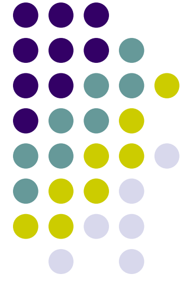
- Two types of communication:
  - Blocking:
    - Safe to change content of buffer holding on to data in the MPI send call
  - Non-blocking:
    - Be careful with the data in the buffer, since you might step on/use it too soon

- MPI provides four modes for these two types
  - standard, synchronous, buffered, ready



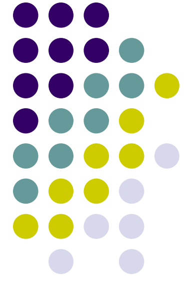
Collective Actions

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

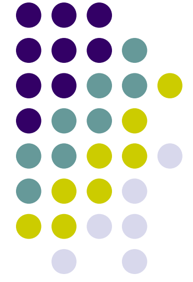


- Syntax:

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

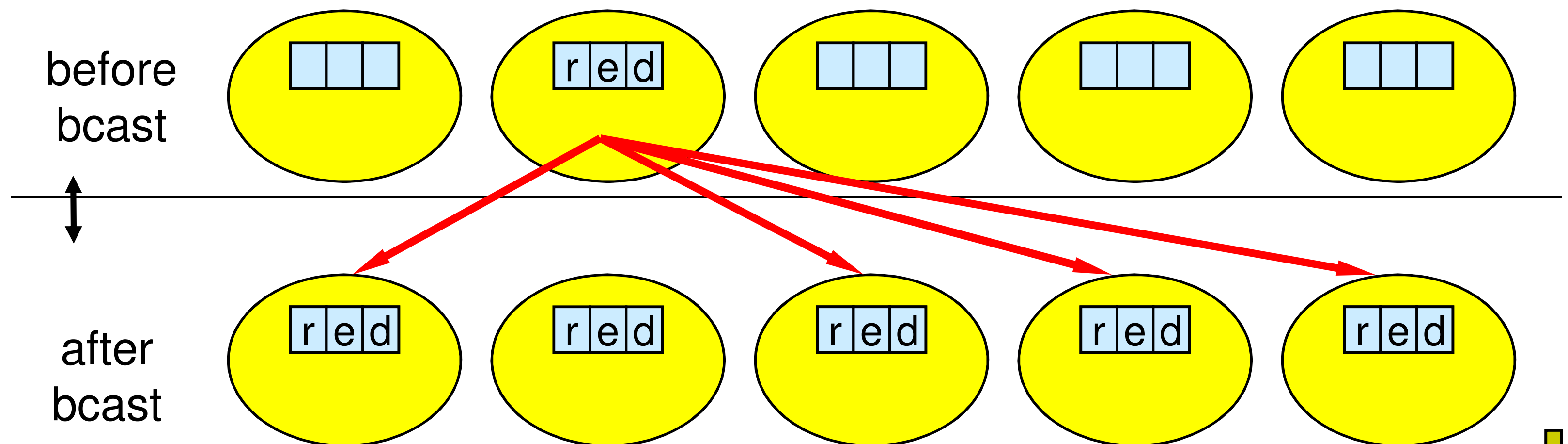
- `MPI_Barrier` not needed that often:
  - All synchronization is done automatically by the data communication
    - A process cannot continue before it has the data that it needs
  - If used for debugging
    - Remember to remove for production release

Communication Action: Broadcast



- Function prototype:

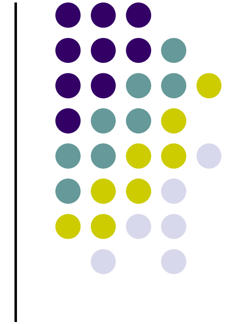
```
int MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```



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

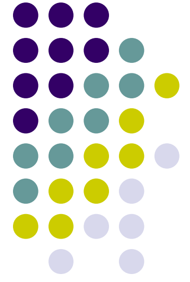
[ICHEC]→

MPI_Bcast



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

MPI_Bcast

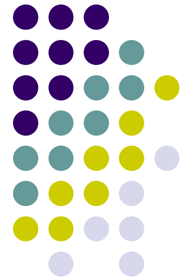


```
int MPI_Bcast (void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm);
```

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

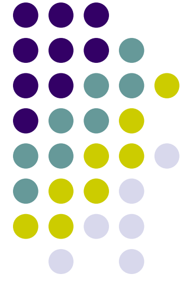


- Read a parameter file on a single processor and send data to all processes

```
#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();
}
```

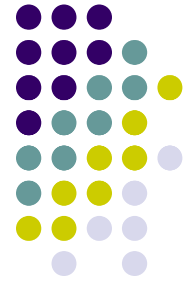
Example: MPI_Bcast

[Output]



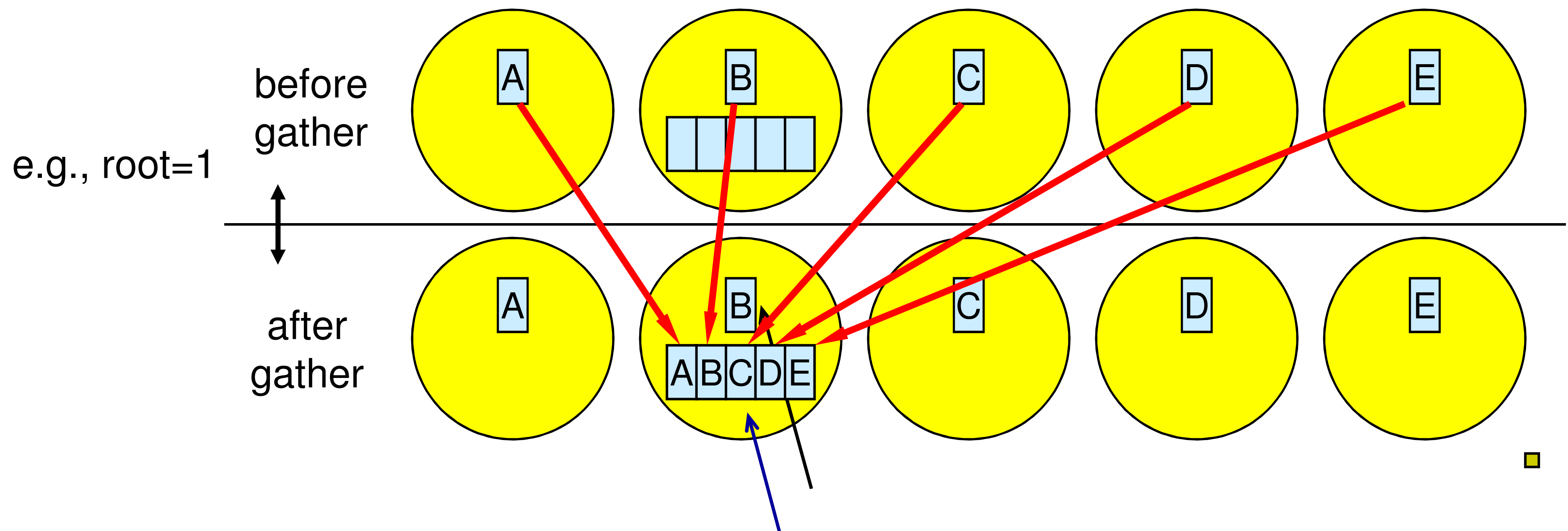
```
[negrut@euler CodeBits]$ qsub -I -l nodes=8:ppn=4,walltime=5:00
qsub: waiting for job 16114.euler to start
qsub: job 16114.euler ready

[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



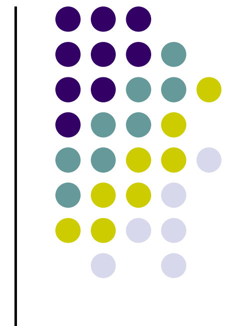
- Function Prototype

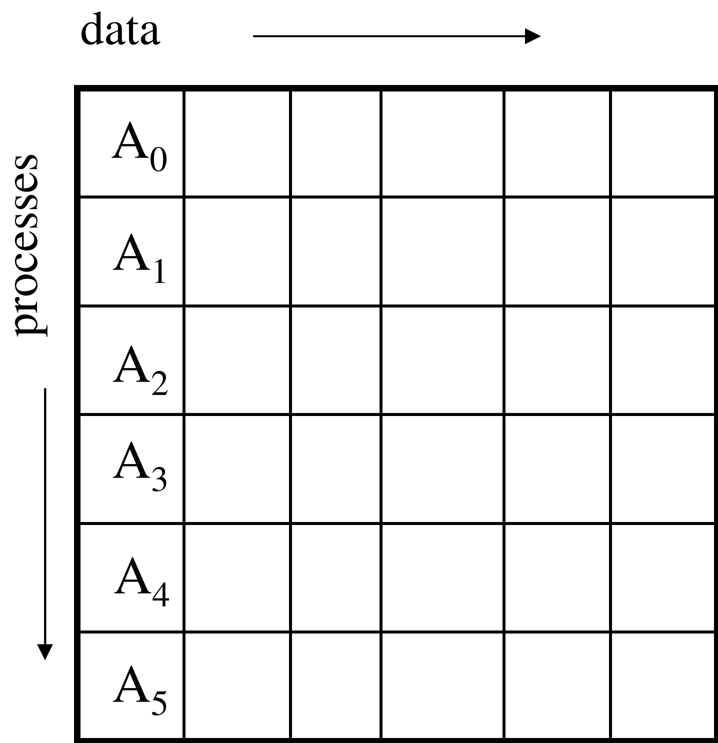
```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



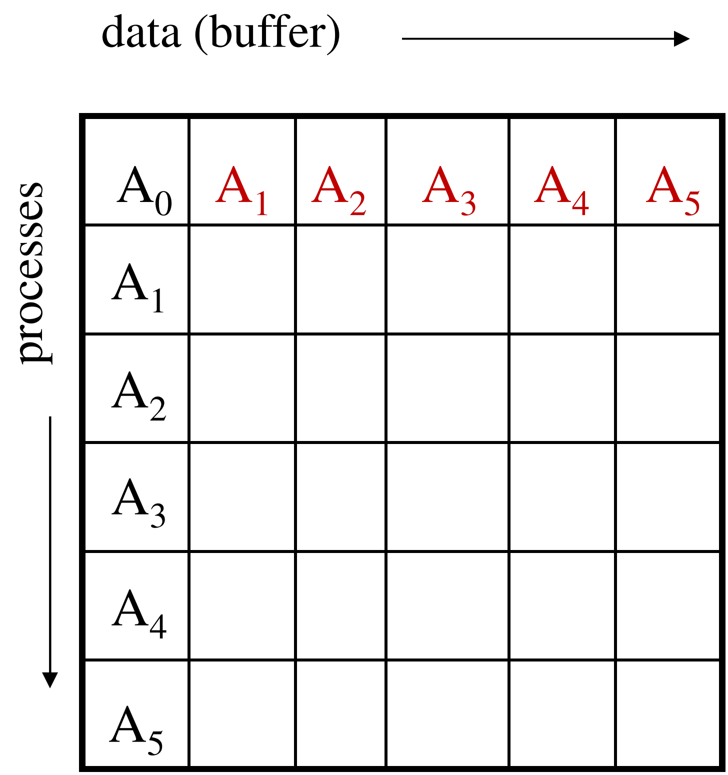
[ICHEC]→

MPI_Gather



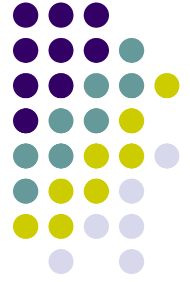


Gather



[A. Siegel]→

MPI_Gather

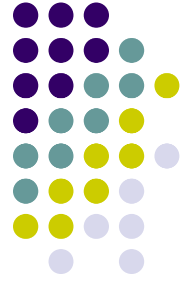


```
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- 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)

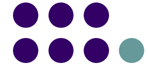
[A. Siegel]→

MPI_Gather



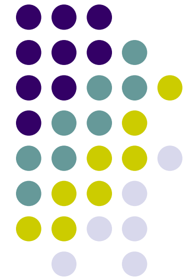
- Each process sends content of send buffer to the root process
- Root receives and stores in rank order
- Remarks:
  - Receive buffer argument ignored for all non-root processes (also recvtype, etc.)
  - `recvcount` on root indicates number of items received from each process, not total. This is a very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

[A. Siegel]→



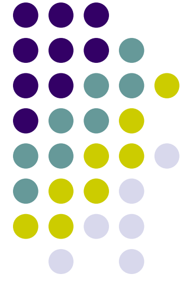
```c
#include "mpi.h"
#include <stdlib.h>
int main(int argc, char **argv){
    int myRank, nprocs, nlcl=2, n, i;
    float *data, *data_loc;

    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;
}
```

[A. Siegel]→



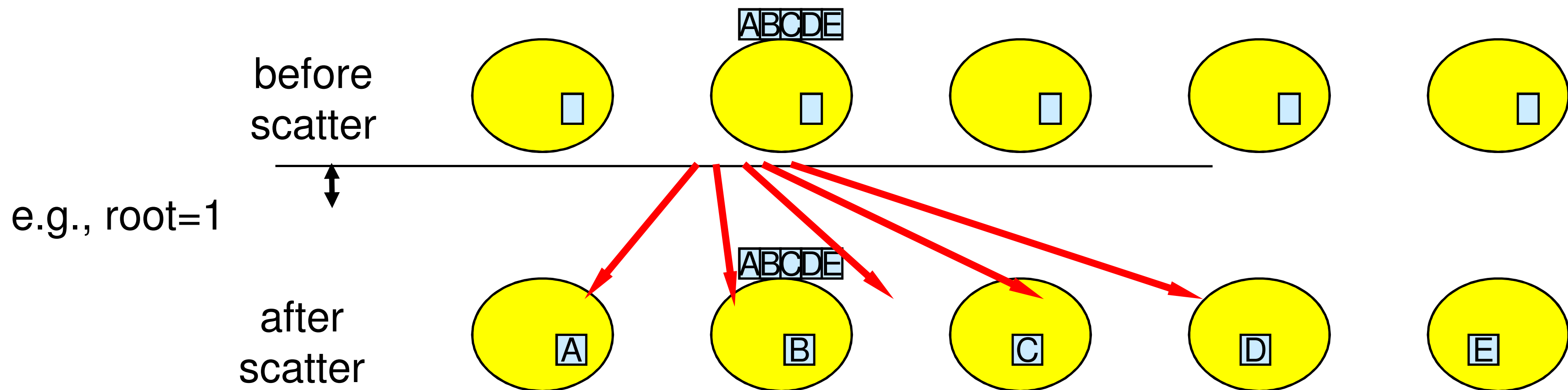
```
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
0.000000
1.000000
1.000000
2.000000
2.000000
3.000000
3.000000
4.000000
4.000000
5.000000
5.000000
[negrut@euler20 CodeBits]$
```

Communication Action: Scatter



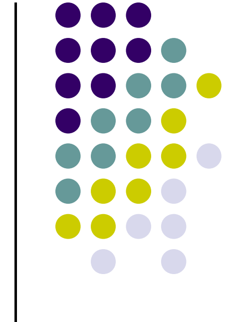
- Function prototype

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



[ICHEC]→

MPI_Scatter



data (buffer) →

processes ↓

A_0	A_1	A_2	A_3	A_4	A_5

Scatter →

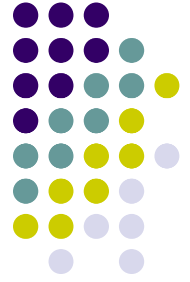
data (buffer) →

processes ↓

A_0	A_1	A_2	A_3	A_4	A_5
A_1					
A_2					
A_3					
A_4					
A_5					

[A. Siegel]→

MPI_Scatter

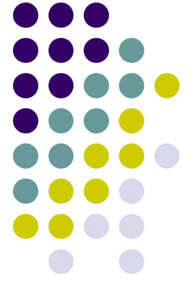


```
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- 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)

[A. Siegel]→

MPI_Scatter



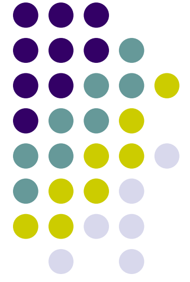
- Inverse of MPI_Gather
- Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter
- Remarks:
  - All arguments are significant on **root**, while on other processes only **recvbuf**, **recvcount**, **recvtype**, **root**, and **comm** are significant

[A. Siegel]→

```c
#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;
}
```

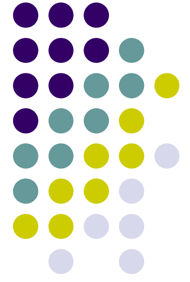
This is interesting. Think what's happening here…

[A. Siegel]→



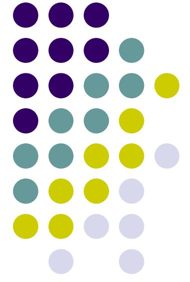
```
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
10.000000
11.000000
[negrut@euler20 CodeBits]$
```

Putting Things in Perspective…



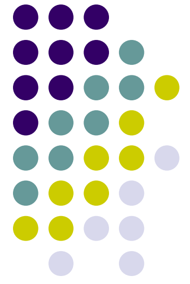
- Gather: you automatically create a serial array from a distributed one
- Scatter: you automatically create a distributed array from a serial one

Global Reduction Operations



- To perform a global reduce operation across all members of a group.
- d_0 **o** d_1 **o** d_2 **o** d_3 **o** … **o** d_{s-2} **o** d_{s-1}
  - d_i = data in process rank i
    - single variable, or
    - vector
  - **o** = 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 **o** d_1) **o** (d_2 **o** d_3)] **o** [… **o** (d_{s-2} **o** d_{s-1})]
  - (((((d_0 **o** d_1) **o** d_2) **o** d_3) **o** …) **o** d_{s-2}) **o** d_{s-1})

Example of Global Reduction

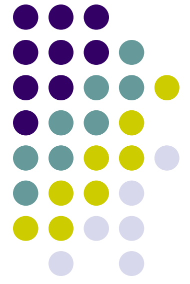


- Global integer sum
- Sum of all `inbuf` values should be returned in `resultbuf`.
- Assume `root`=0;

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

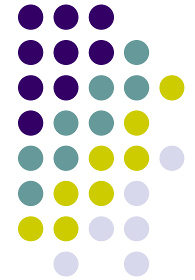
- The result is only placed in `resultbuf` at the root process.

Predefined Reduction Operation Handles



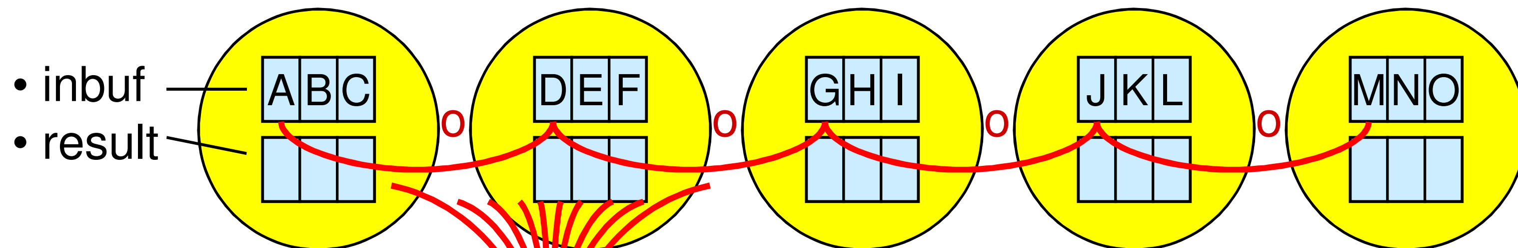
Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum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

MPI_Reduce

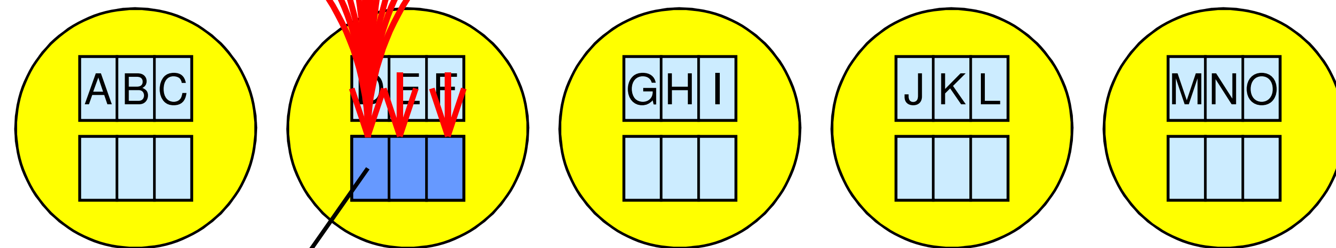


before MPI_REDUCE

- inbuf
- result



after



root=1

AoDoGoJoM

[ICHEC]→