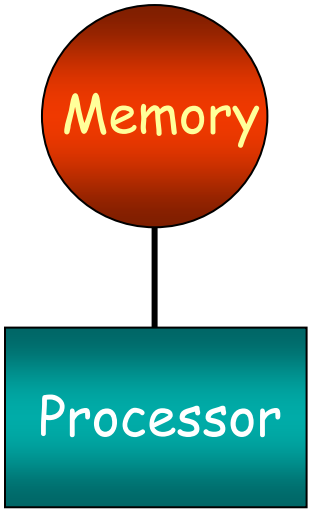
Message-Passing Programming Paradigm

S.S. Kadam
C-DAC, Pune
sskadam@cdac.in

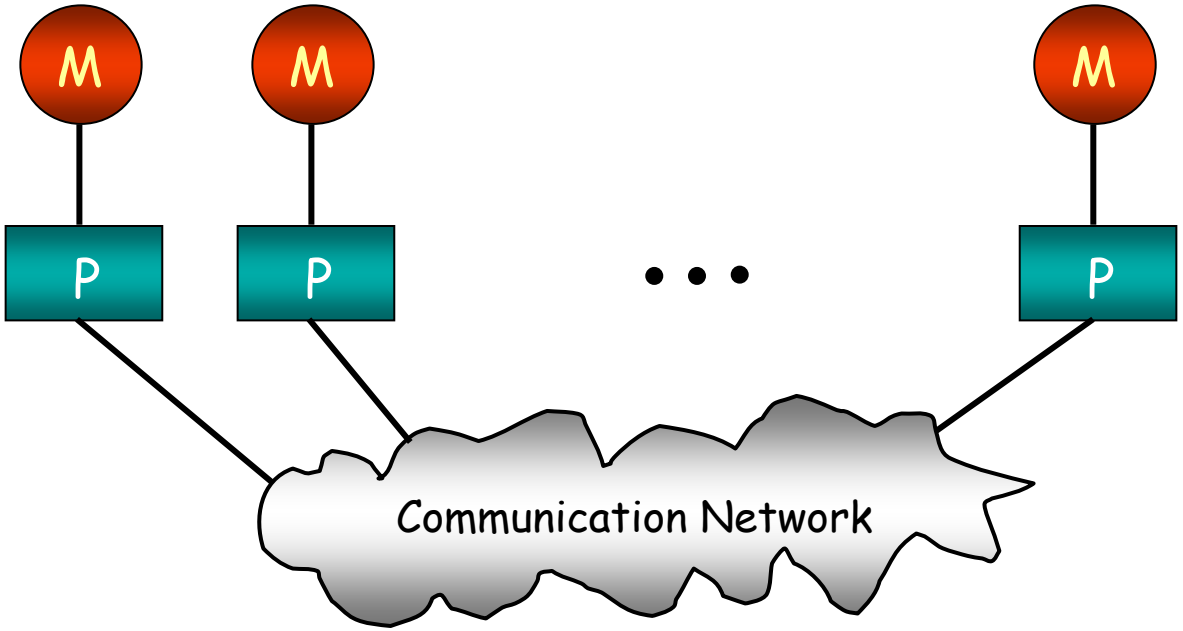
Sequential Programming Paradigm

- Programmer has a simplified view of the target machine
- Single processor has access certain amount of memory
- May be implemented in *time-sharing* environment, where other processes share the processor and memory
  - But individual programs can map to any sequential architecture



Message-Passing Programming Paradigm

- Many instances of sequential paradigm considered together
- Programmer imagines several processors, each with own memory, and writes a program to run on each processor
- Processes communicate by sending messages to each other



Message-Passing Platform

- Has no concept of a shared memory space or of processors accessing each other's memory directly
- However, programs written in message-passing style can run on any architecture that supports such model such as
  - Distributed or shared-memory multi-processors
  - Networks of workstations
  - Single processor systems

Message Transfer

- Occurs when data moves from variables in one sub-program to variables in another sub-program
- Sending and receiving processors cooperate to provide required information for message transfer

Message Information

- Message passing system provides following information to specify the message transfer
  - Which processor is sending the message
  - Where is the data on the sending processor
  - What kind of data is being sent
  - How much data is there
  - Which processor(s) are receiving the message
  - Where should the data be left on the receiving processor
  - How much data is receiving processor prepared to accept

Progress of Message Communication

- Receiving processor should be aware of incoming data
- Sending processor should know about delivery of its message
- Message transfer also provides *synchronization* information in addition to data in the message

Access

- Before messages can be sent a sub-program needs to be connected to the message passing system
- Some message passing systems allow processors to have multiple connections to message passing system
- Others support only single connection
  - Mechanisms needed to distinguish diff. types of messages

Address

- Each message must be addressed
  - Should contain *envelope* information (containing address) that can also be accessed by the receiving process

Reception

- Receiving process should be capable of dealing with sent messages
  - Buffer should be large enough to hold the data
    - Otherwise message may be truncated or discarded
- Buffers may be
  - variables declared within the application code, or
  - internal to the message passing system

Point to Point Communication

- Simplest form of message communication
- Message is sent from a sending processor to a receiving processor
  - Only these two processors need to know anything about the message

Communication Types

- Several variations exist on how sending of a message influences execution of the sub-program
  - First common distinction is between *synchronous* and *asynchronous* sends
  - Other important distinction is *blocking* and *non-blocking*

Synchronous Send

- Communication does not complete until the message has been received

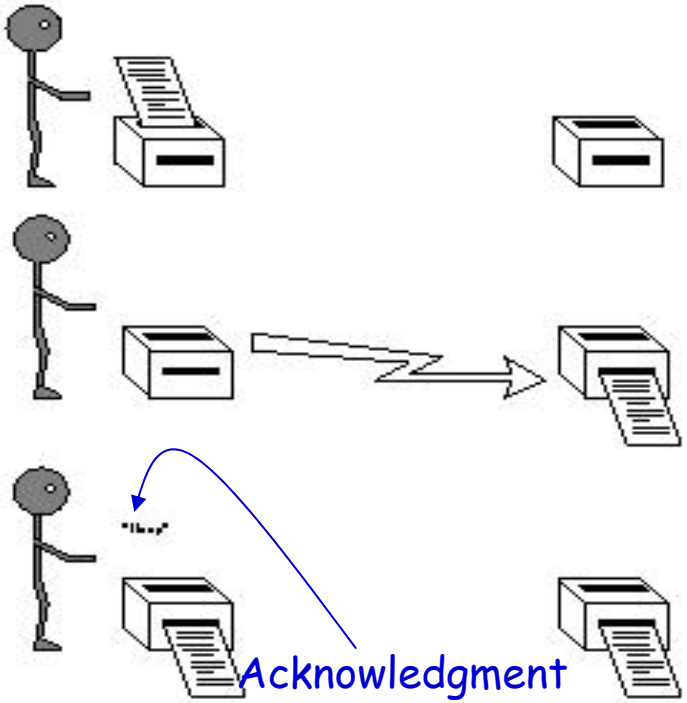


Image taken from http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.book_10.html

Asynchronous Send

- Communication completes as soon as message is on its way
  - *Asynchronous* sends only know when the message has left

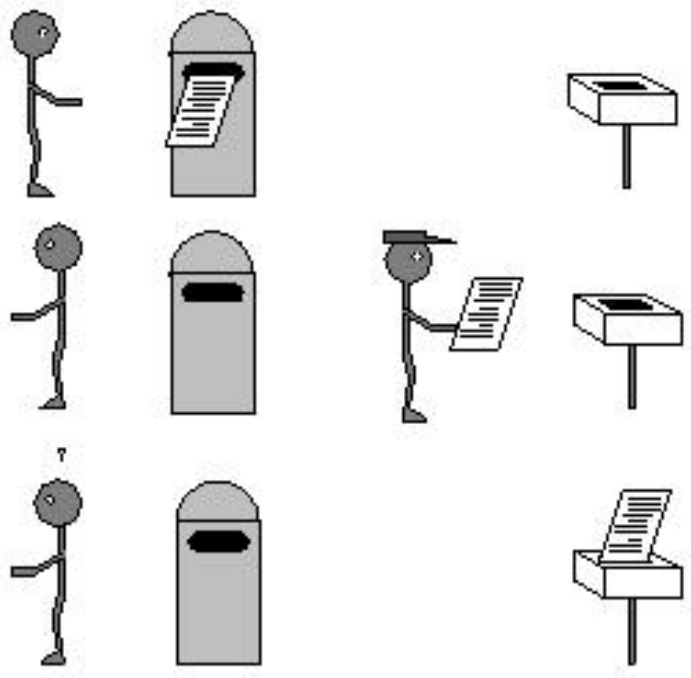
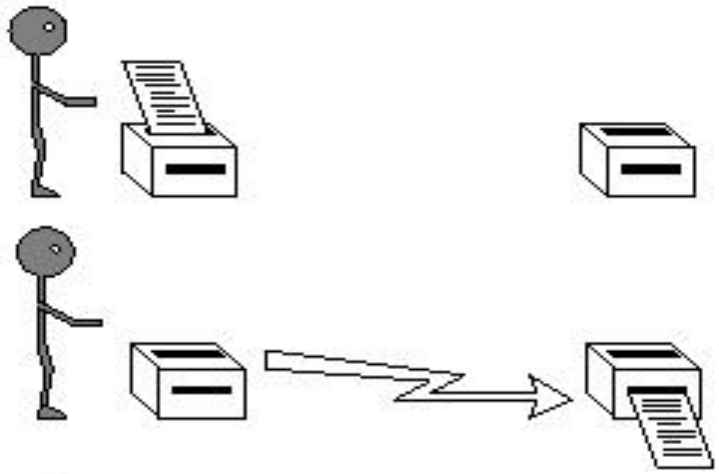


Image taken from http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.book_10.html

Blocking Operation

- Programs return from the subroutine call when the local transfer operation has completed
  - Though message transfer may not have been completed



Q: Does he wait for the message to be received?

Image taken from http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.book_10.html

Non-blocking Operation

- Returns straight away, allows sub-program to perform useful work while waiting for communication to complete
- Sub-program can later test for completion of operation

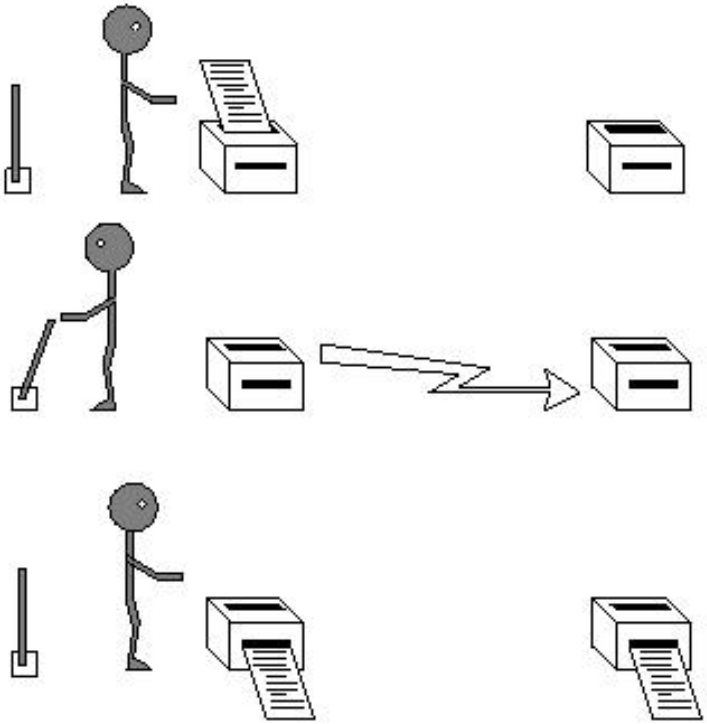


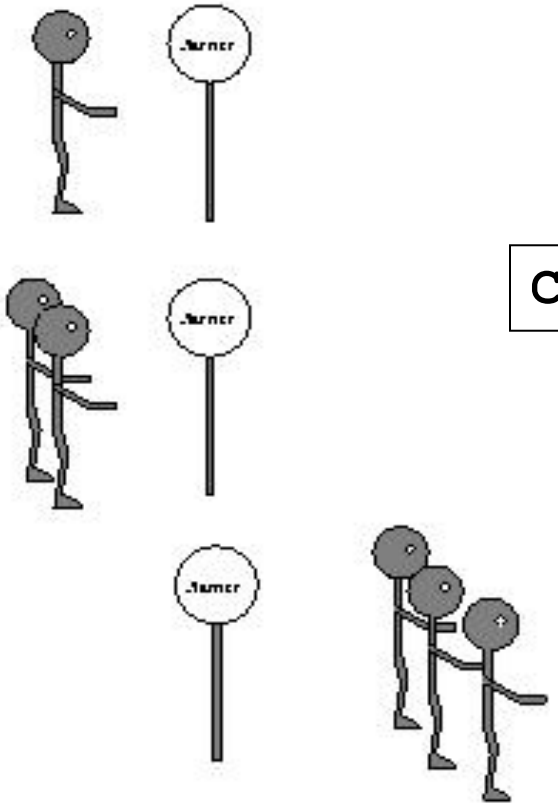
Image taken from http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.book_10.html

Collective Communications

- Point-to-point communication operations discussed until now involve a pair of communicating processes
- Many message-passing systems also provide operations which allow larger number of processes to communicate

Barrier

- A barrier operation synchronizes a number of processors
- No data is exchanged but the barrier blocks until all participating processes have called the barrier routine



```
Call MPI_Barrier(MPI_Comm,Err)
```

Call to `MPI_Barrier` returns only after all the processes in the group have called this function

Image taken from http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.book_12.html

Broadcast

- A broadcast is a one-to-many communication operation
- One process sends the same message to several destination processes with a single operation

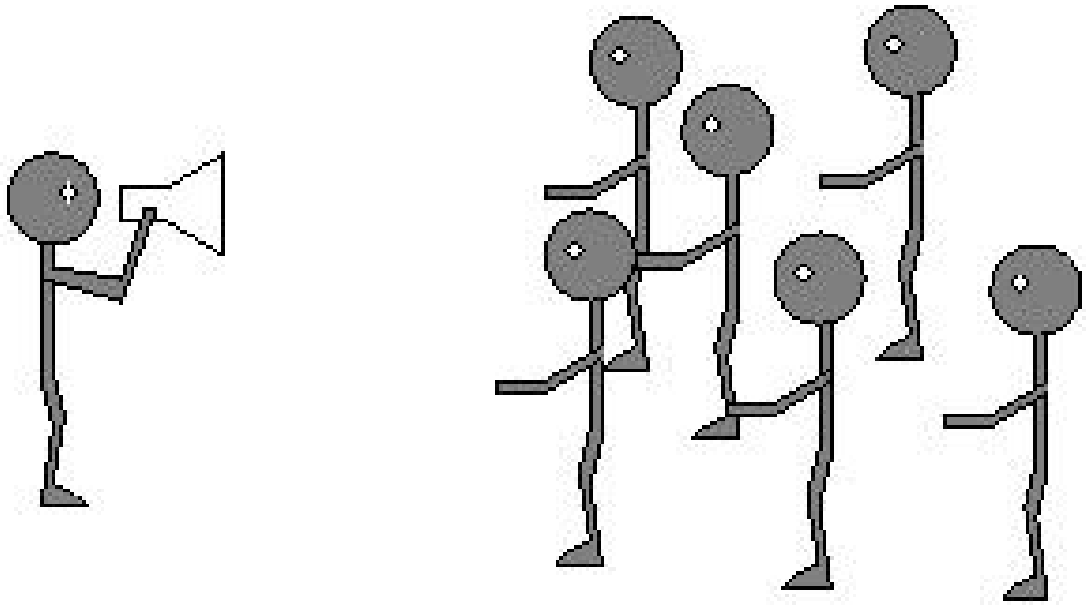
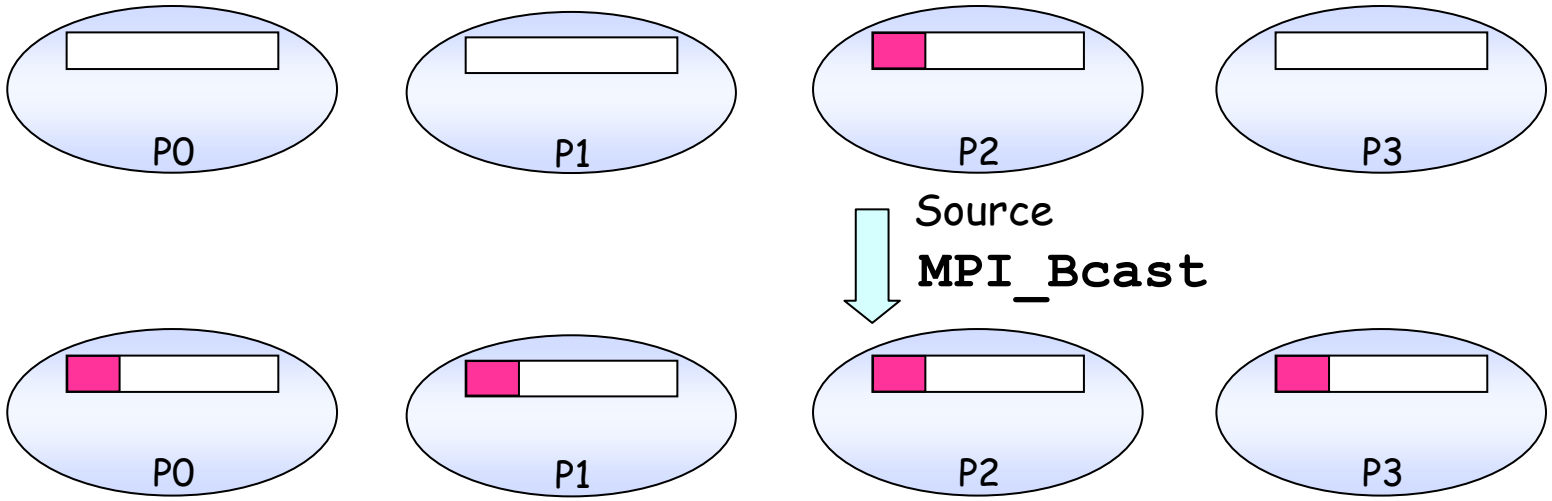


Image taken from http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.book_13.html

Broadcast (conti.)

```
Call MPI_Bcast(buf, count, datatype, source, MPI_Comm, Err)
```

- Sends data stored in buffer `buf` of process `source` to all the other processes in the group `comm`

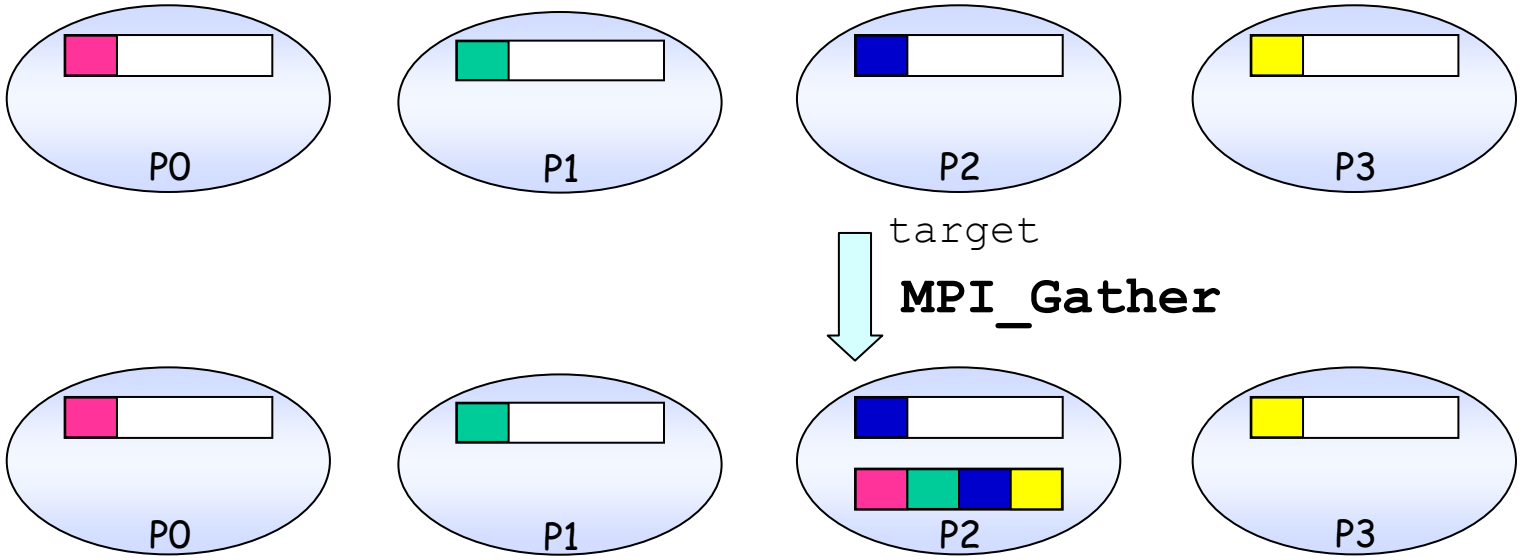


Note: ■ contains `count` * `datatype` elements

Gather

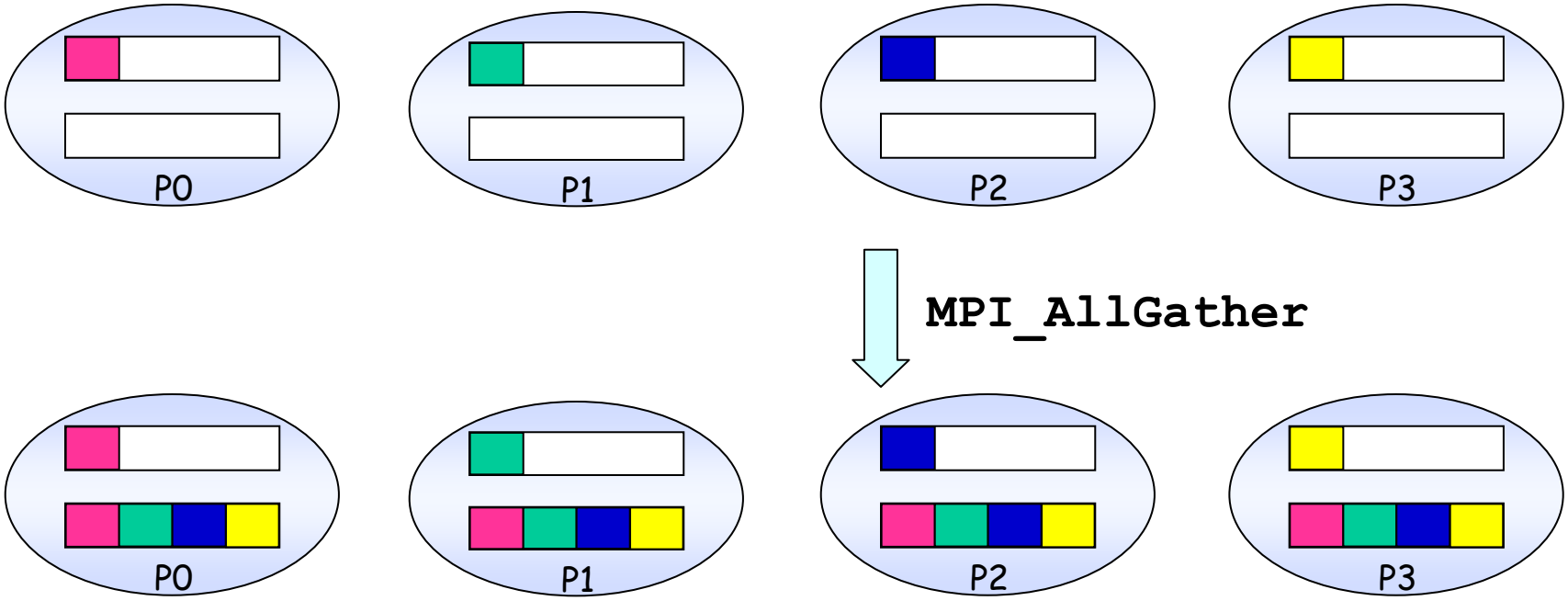
```
Call MPI_Gather(sendbuf, sendcount, senddatatype,
recvbuf, recvcount, recvdatatype, target, MPI_comm, Err)
```

- Each process (incl. `target`), sends data stored in array `sendbuf`, to the `target` process
  - `recvbuf` of target process stores data in rank order
  - `recvcount` specifies no. of elements received from each process



AllGather

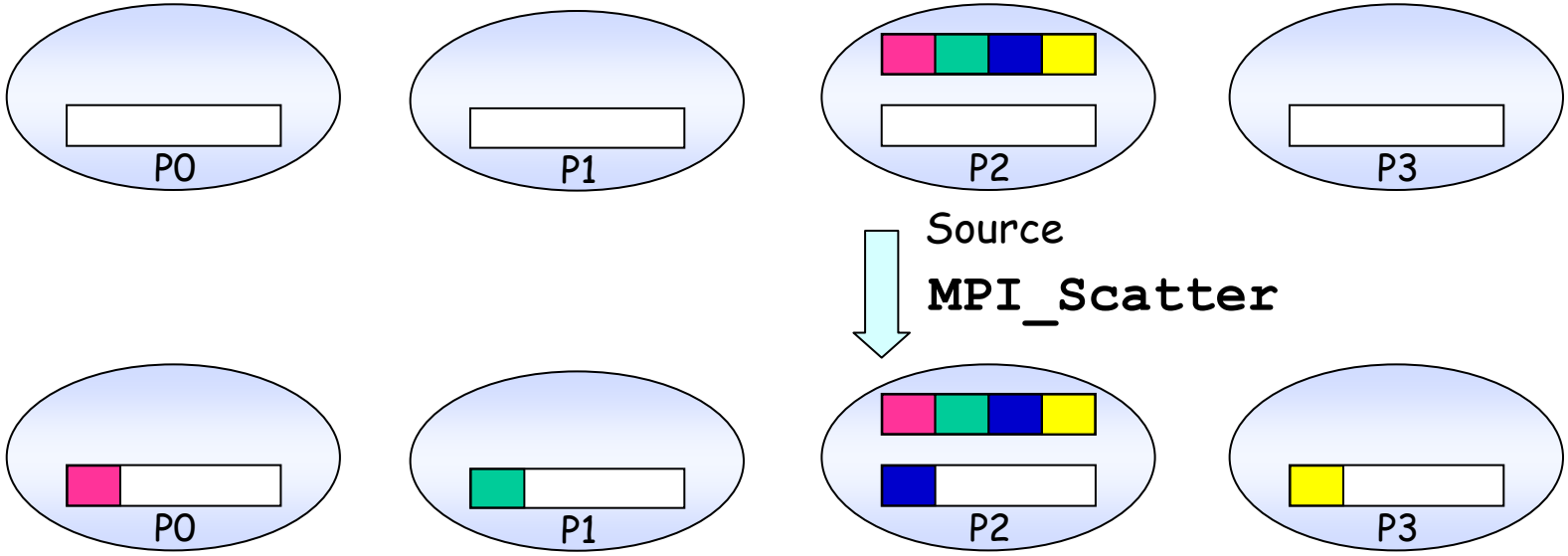
```
Call MPI_Allgather(sendbuf, sendcount, senddatatype,
recvbuf, recvcount, recvdatatype, MPI_comm)
```



Scatter

```
Call MPI_Scatter(sendbuf, sendcount, senddatatype,
recvbuf, recvcount, recvdatatype, source, MPI_Comm)
```

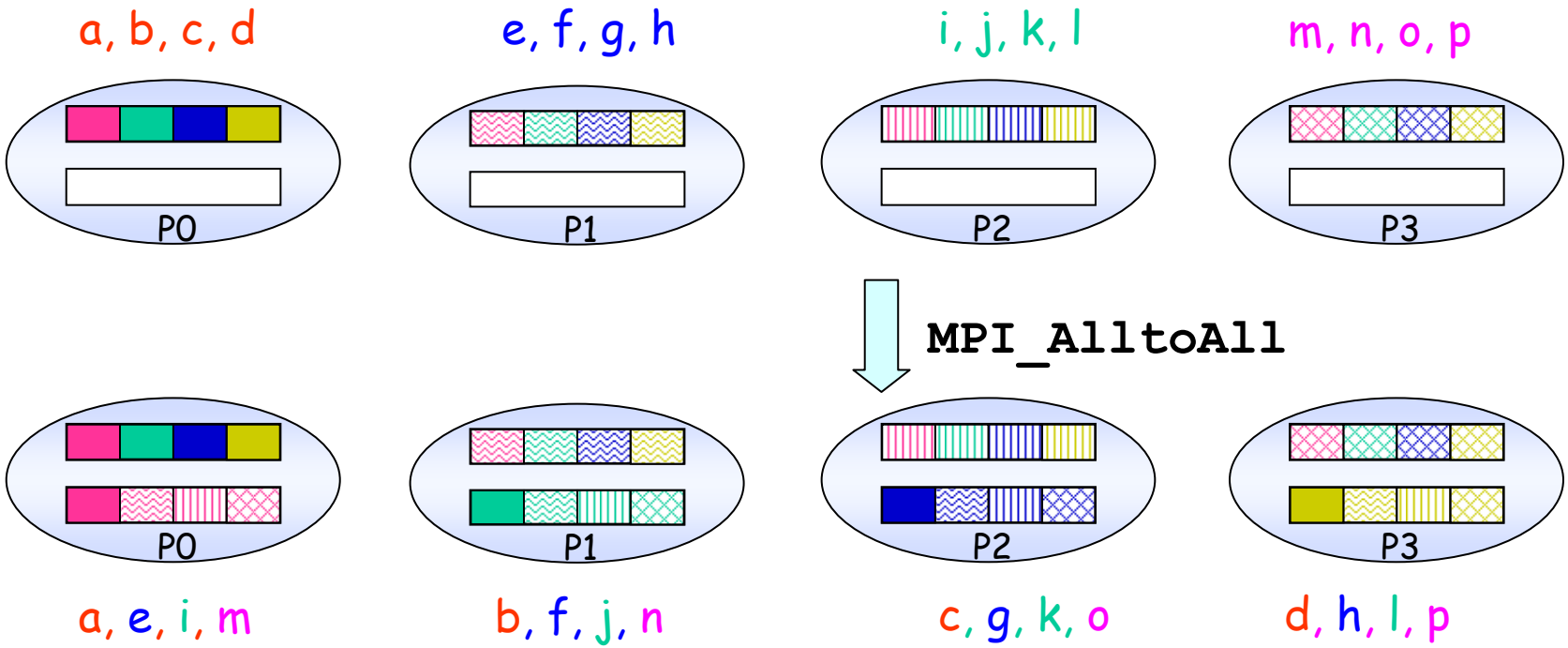
- The `source` process sends different part of `sendbuf`, to each process (incl. itself)
  - `recvbuf` of target process stores data in rank order
  - `sendcount` specifies no. of elements sent to each process



All-to-All

```
Call MPI_AlltoAll(sendbuf, sendcount, senddatatype,
recvbuf, recvcount, recvdatatype, source, MPI_Comm)
```

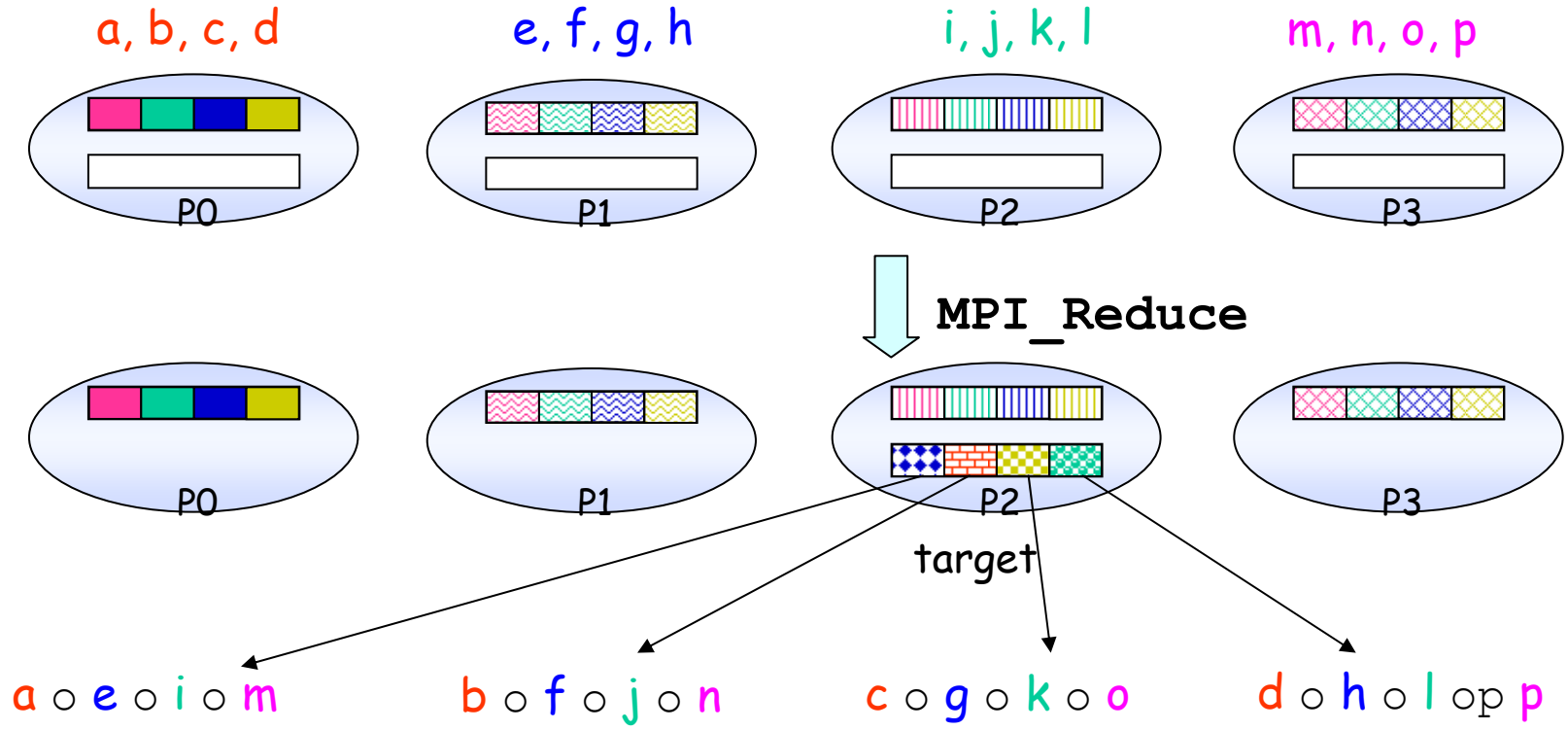
- Each process sends a different portion of `sendbuf` to each other process (incl. itself)
  - `recvbuf` of target process stores data in rank order
  - `sendcount` specifies no. of elements sent to each process



Reduction

```
Call MPI_Reduce(sendbuf, recvbuf, count, datatype,
MPI_Op, target, MPI_Comm)
```

- Combines elements in `sendbuf` of each process in group using MPI_op operation, and returns combined values in `recvbuf` of `target` process
- All processes must call `MPI_Reduce` with same value for `count`



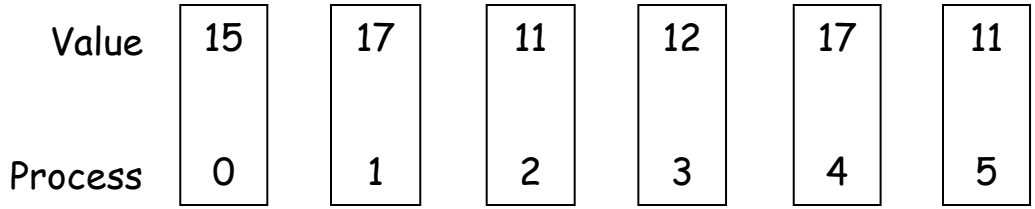
Reduction (conti.)

```
Call MPI_AllReduce(sendbuf, recvbuf, count, Datatype,
MPI_Op, MPI_Comm)
```

- All processes receive the result of the operation, hence, there is no target argument

Some Predefined reduction operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	integers and floating point
MPI_MIN	Minimum	integers and floating point
MPI_SUM	Sum	integers and floating point
MPI_PROD	Product	integers and floating point
MPI_LAND	Logical AND	integers
MPI_BAND	Bit-wise AND	integers and byte
. . .	. . .	. . .
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs



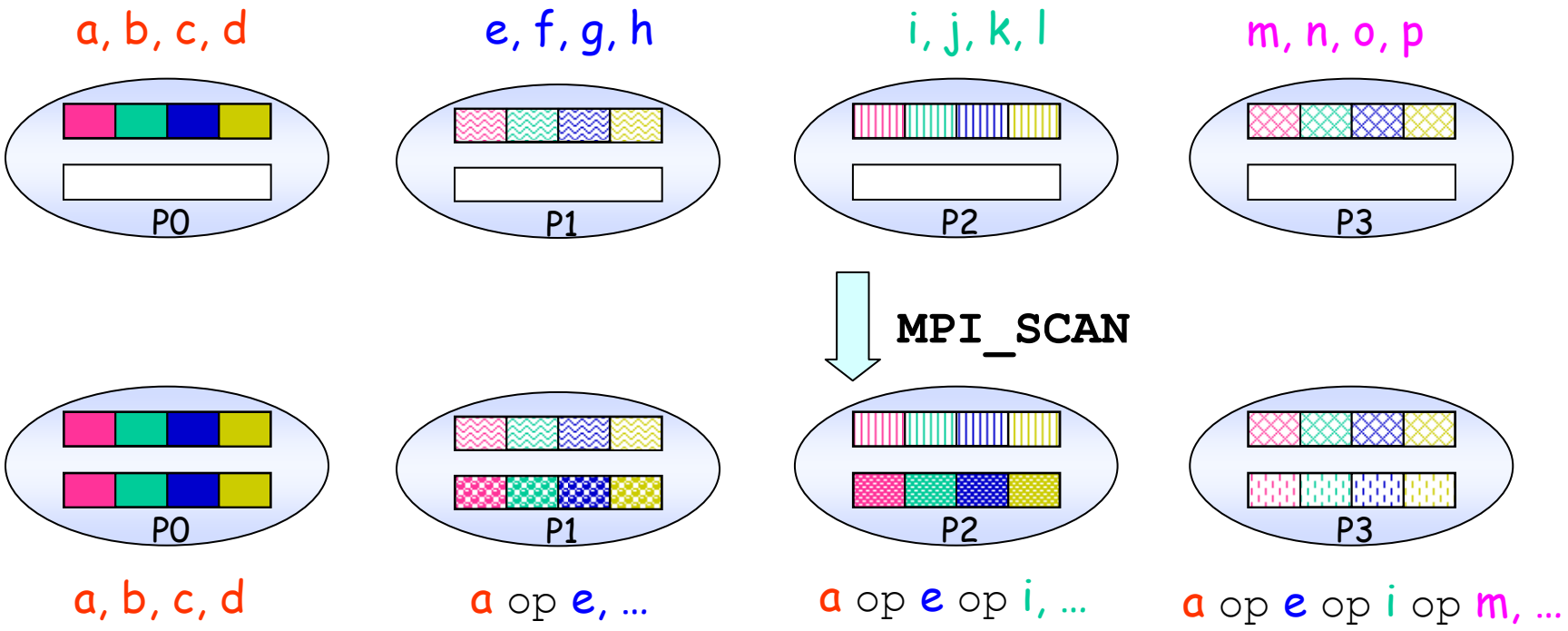
MinLoc(Value, Process) = (11, 2) *(Min value, Min processor ID)*

MaxLoc(Value, Process) = (17, 1) *(Max value, Min processor ID)*

Prefix

`int MPI_Scan(sendbuf, recvbuf, count, datatype, MPI_Op op, MPI_Comm)`

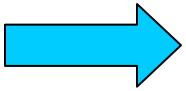
- Performs prefix reduction of data in `sendbuf` buffer at each process and returns the result in buffer `recvbuf`



Variations

- `MPI_Gatherv(...)`, `MPI_Allgatherv(...)`
  - Allow different number of data elements to be sent by each process
    - Parameter `recvcount` replaced with array `recvcounts`
    - Additionally, another array parameter determines where in `recvbuf` the data sent by each process will be stored
- `MPI_Scatterv(...)`
  - Allows different amounts of data to be sent to different processes
    - Parameter `sendcount` replaced with array `sendcounts`
    - Additionally, another array parameter determines where in `sendbuf` these elements will be sent from
- `MPI_AlltoAllv(...)`
  - Allows different amounts of data to be sent to and received from each process

Summary

P0	P1	P2*	P3	Function	P0	P1	P2	P3
a	b	c	d	MPI_Gather			a,b,c,d	
a	b	c	d	MPI_Allgather	a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
		a,b,c,d		MPI_Scatter	a	b	c	d
a,b,c,d	e,f,g,h	i,j,k,l	m,n,o,p	MPI_AlltoAll	a,e,i,m	b,f,j,n	c,g,k,o	d,h,l,p
		b		MPI_Bcast	b	b	b	b
SBuf	SBuf	SBuf	SBuf		RBuf	RBuf	RBuf	RBuf

* Sender/Root process required by MPI_Gather, MPI_Scatter, MPI_Bcast

Self Revision

- Browse http://www.abo.fi/~mats/HPC1999/examples/
  - **Copy, Paste, and Execute at least 6 programs spawning**
    - **Blocking Send**
    - **Buffered Send**
    - **Non-Blocking Send**
    - **Collective Communications (MPI_Bcast, MPI_Scatter, MPI_Gather, etc.)**
  - **Experiment with deadlocks**