

Message Passing

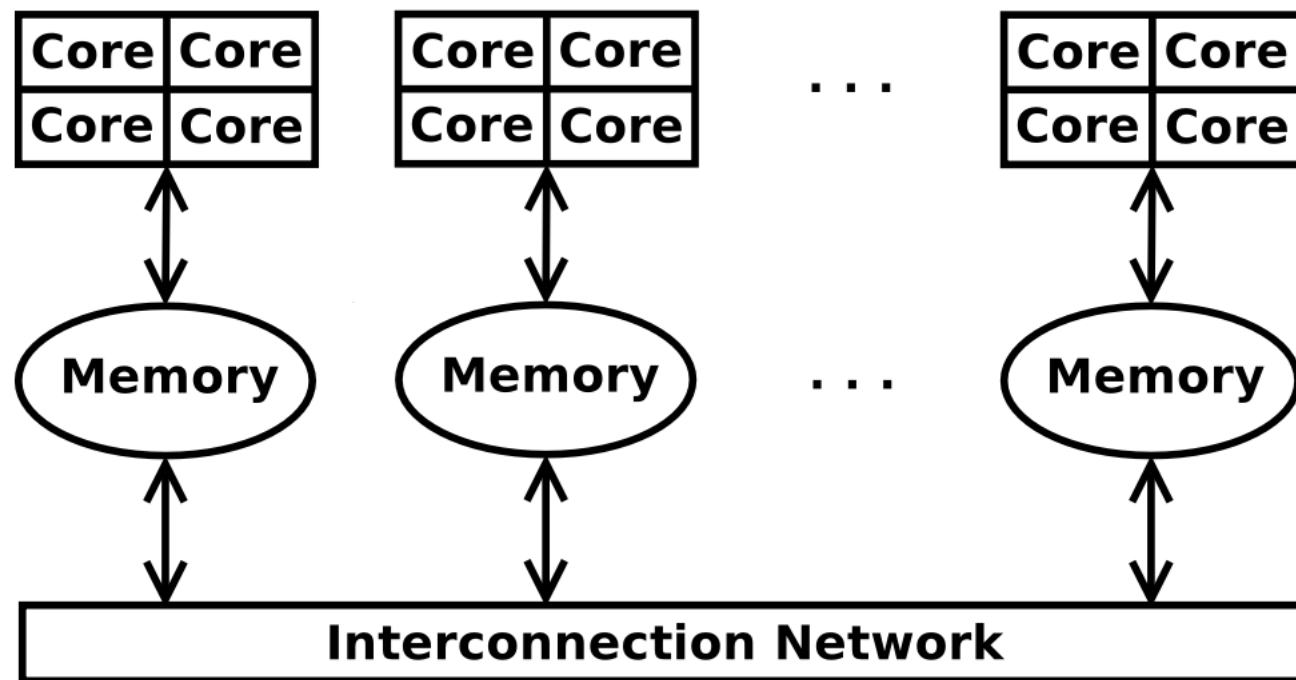
Academic year 2023-2024

Prof. Marco Aldinucci, Prof. Massimo Torquati

Outline

- 1) Execution models
- 2) A taxonomy of communications
 - Synchronous/asynchronous – Symmetric/asymmetric
- 3) Tags, Communicators
- 4) From the abstract model to MPI
- 5) Execution

Abstract execution model



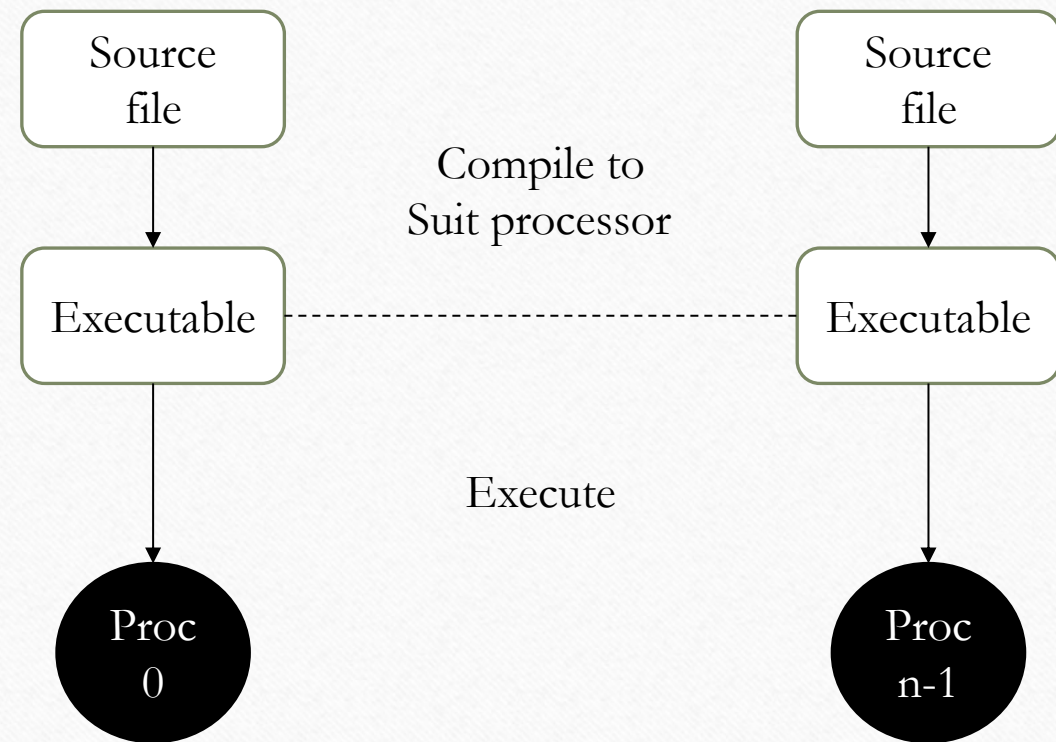
User-level Message-Passing Programming

Two primary mechanisms are needed:

1. A method of creating separate processes for execution on different computers
 - MPMD, SPMD
2. A method of sending and receiving messages

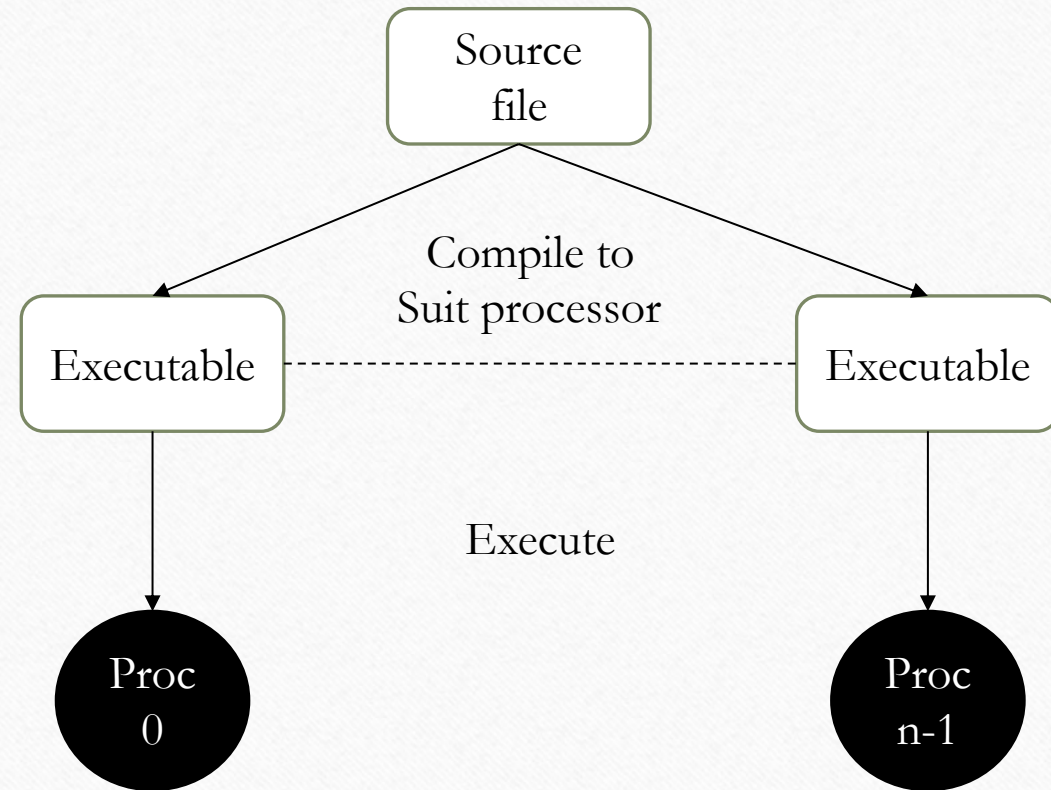
Multiple Program, Multiple Data (MPMD) model

- Separate programs for each processor.



Single Program Multiple Data (SPMD) model

- Different processes merged into one program. Control statements select different parts for each processor to execute. All executables started together - *static process* creation
- (Basic MPI way)



MPI Helloworld

- Interface with multiple implementations
- MPI-1 1994, currently MPI-3
- Basic concepts
 - Process creation
 - Purposely not defined - depends upon the implementation
 - Init and Finalize
 - Communicator and Rank
 - Defines the scope of a communication operation
 - Processes have ranks associated with communication.
 - Initially, all processes enrolled in MPI_COMM_WORLD
 - Other communicators can be established for groups of processes.
 - Communications
 - Point-to-point or collective – sync/async
- Language bindings: C, Fortran, C++

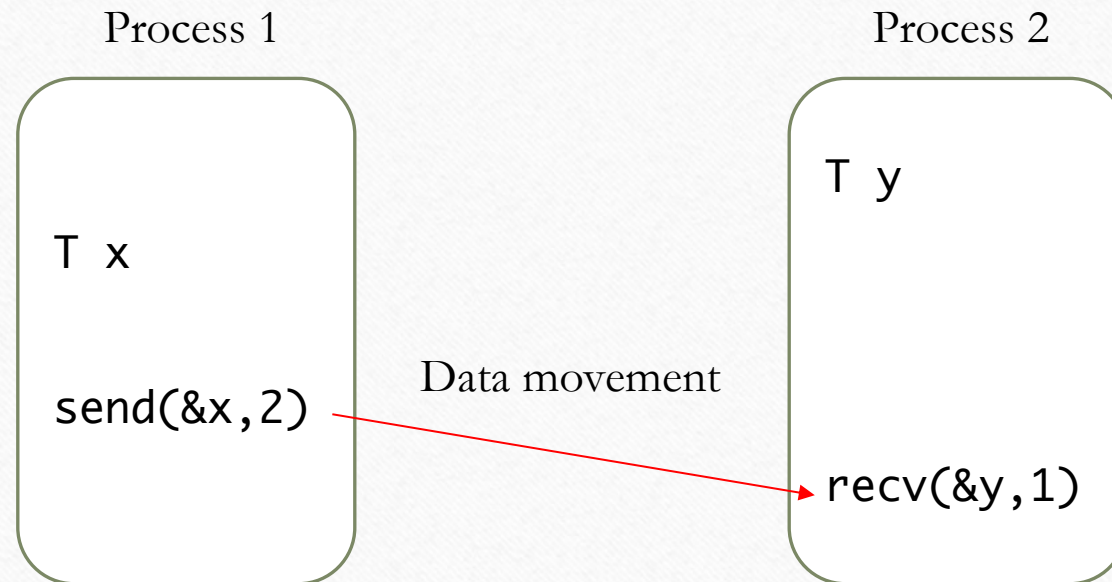
```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <iostream>
4
5  int main (int argc, char *argv[]) {
6      // Initialize MPI
7      MPI_Init(&argc, &argv);
8
9      // Get the number of processes
10     int numP;
11     MPI_Comm_size(MPI_COMM_WORLD, &numP);
12     // Get processor name
13     char processor_name[MPI_MAX_PROCESSOR_NAME];
14     int namelen;
15     MPI_Get_processor_name(processor_name, &namelen);
16     // Get the ID of the process
17     int myId;
18     MPI_Comm_rank(MPI_COMM_WORLD, &myId);
19
20     // Every process prints Hello
21     std::cout << "Process on " << processor_name << " Id " << myId << " of " << numP
22     << ": Hello, world!" << std::endl;
23
24     // Terminate MPI
25     MPI_Finalize();
26     return 0;
27 }
```


Communication Taxonomy

- Symmetric/asymmetric
 - Symmetric or point-to-point or 1:1
 - Asymmetric or collective
 - They are not really only communications but involve some message processing
- Synchronous/asynchronous
 - Synchronous or rendez-vous (w.r.t. comm partners)
 - Asynchronous (w.r.t. comm partners)
 - Blocking (w.r.t. communication channel)
 - Non-blocking (w.r.t. communication channel)

Basic symmetric (point-to-point) Send and Receive Routines

- Passing a message between processes using `send()` and `recv()` library calls:



Synchronous Message Passing

Routines that actually return when message transfer completed.

Synchronous send routine

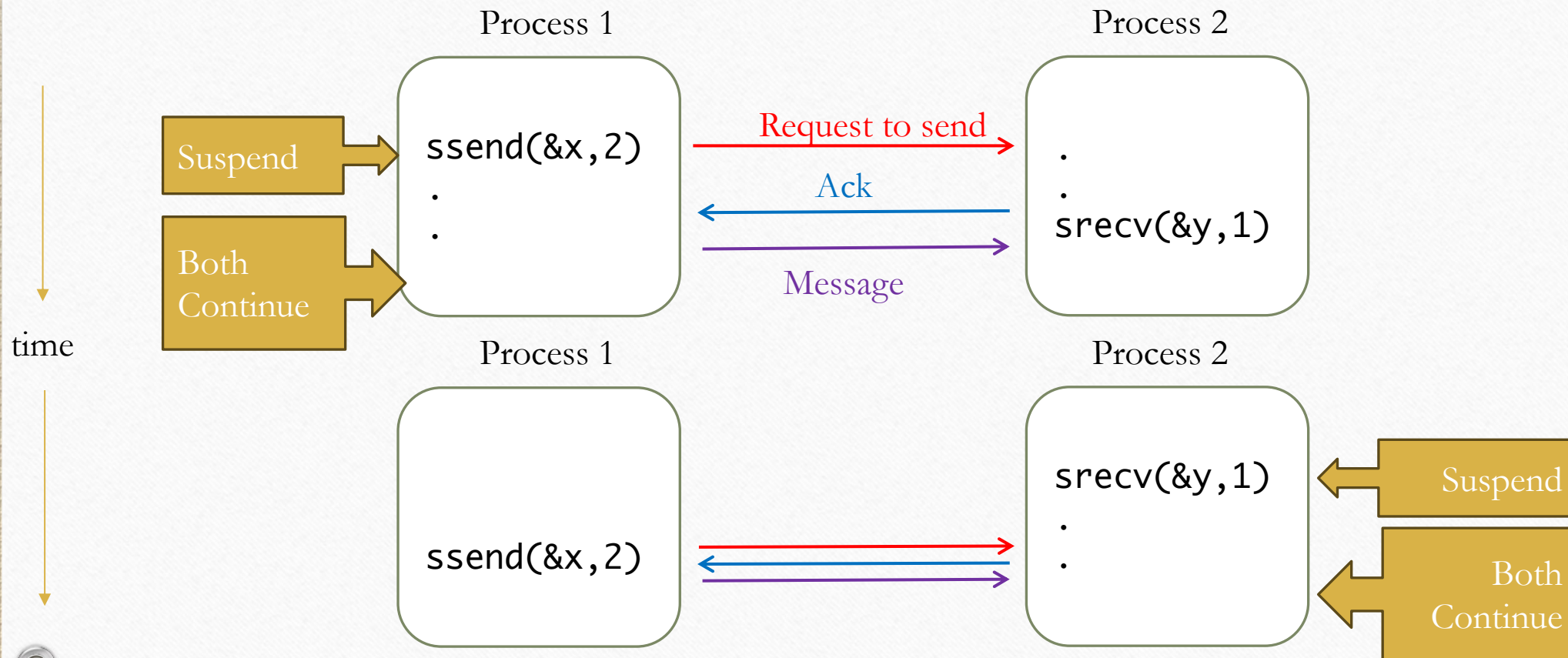
- Waits until complete message can be accepted by the receiving process before sending the message.

Synchronous receive routine

- Waits until the message it is expecting arrives.

Synchronous routines intrinsically perform two actions: They transfer data and they synchronize processes.

Synchronous send() and recv() using 3-way protocol



Asynchronous Message Passing

- Routines that do not wait for actions to complete before returning. Usually require local storage for messages.
- More than one version depending upon the actual semantics for returning.
- In general, they do not synchronize processes but allow processes to move forward sooner. Must be used with care.

MPI Definitions of Blocking and Non-Blocking

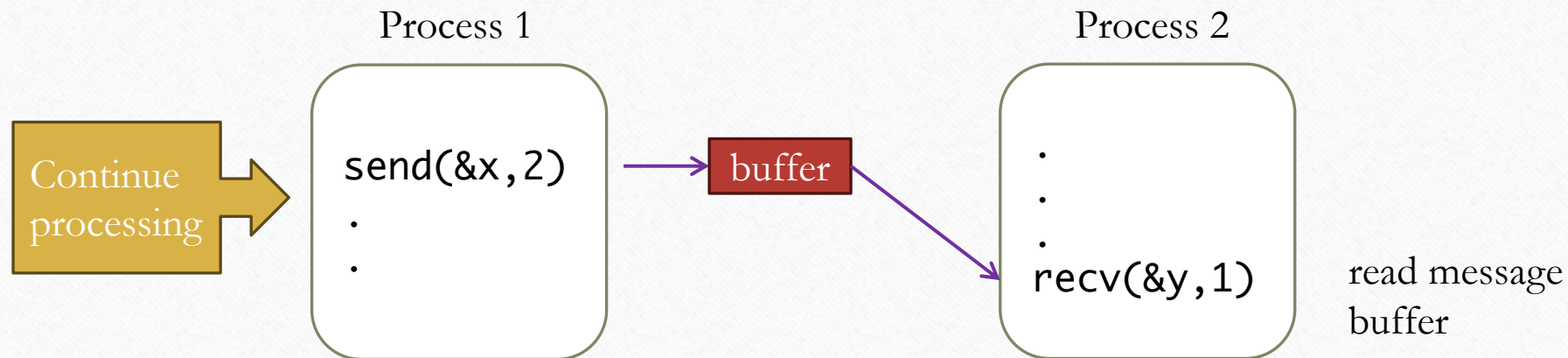
- **Blocking** - return after their local actions complete, though the message transfer may not have been completed.
- **Non-blocking** - return immediately.

Assumes that data storage used for transfer not modified by subsequent statements prior to being used for transfer, and it is left to the programmer to ensure this.

These terms may have different interpretations in other systems.

How message-passing routines return before message transfer completed

- Message buffer needed between source and destination to hold message:



Asynchronous (blocking) routines changing to synchronous routines

- Once local actions are completed and the message is safely on its way, the sending process can continue with subsequent work.
 - Blocking send will send message and return - does not mean that message has been received, just that process free to move on without adversely affecting message.
 - Return when “locally complete” - when location used to hold message can be used again or altered without affecting message being sent.
- Buffers only of finite length and a point could be reached when the send routine held up because all available buffer space is exhausted.
- Then, the send routine will wait until storage becomes re-available - i.e. the routine behaves as a synchronous routine.

Message Tag

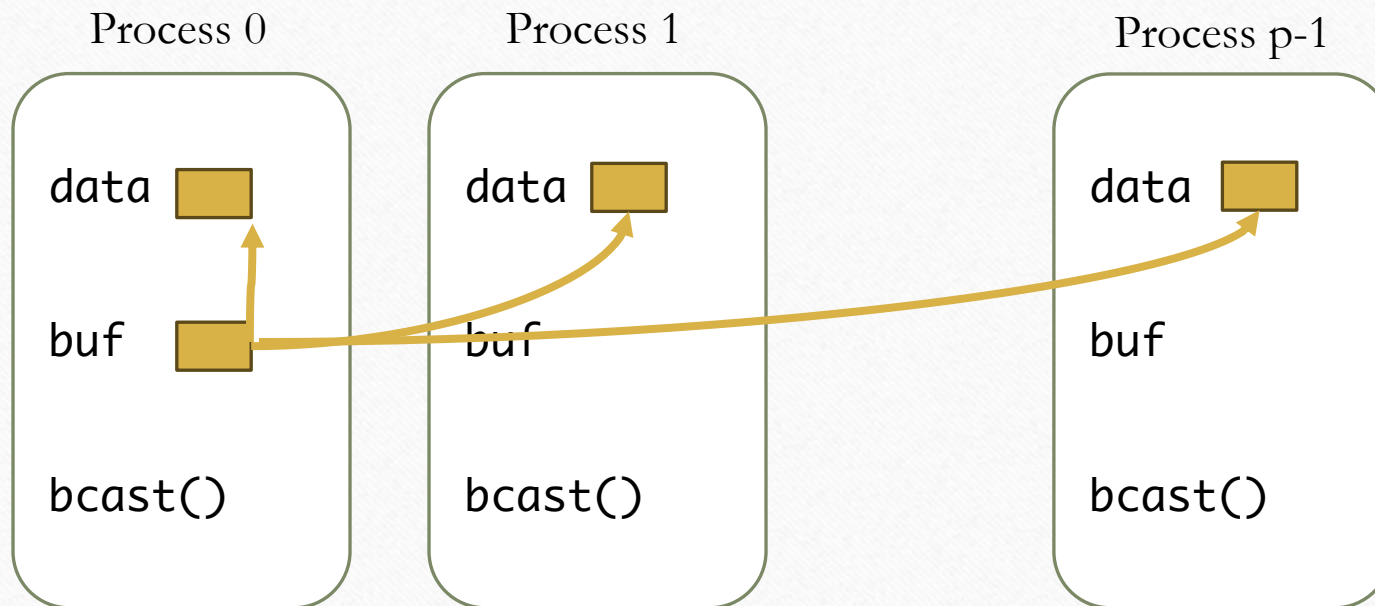
- Used to differentiate between different types of messages being sent
- Message tag is carried within message
- If special type matching is not required, a wild card message tag is used, so that the `recv()` will match with any `send()`

“Group” message passing routines

- Have routines that send message(s) to a group of processes or receive message(s) from a group of processes
- Higher efficiency than separate point-to-point routines although not absolutely necessary
 - They can be simulated with point-to-point

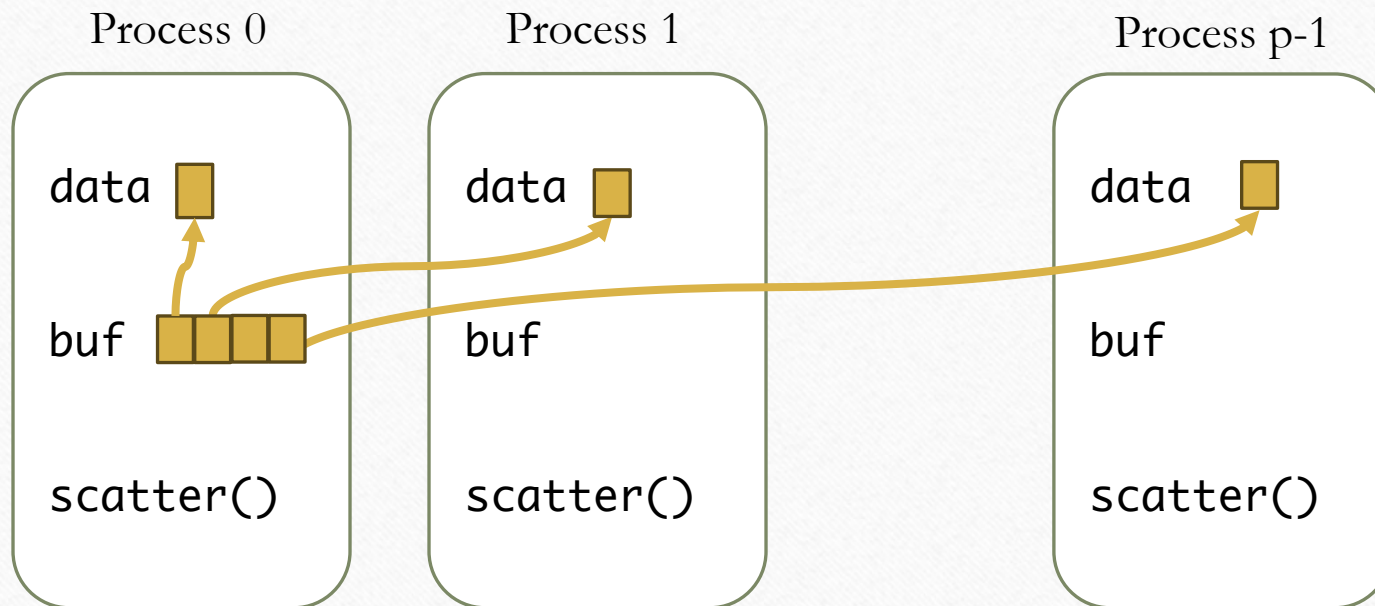
Broadcast

- Sending the same message to all processes concerned with the problem.
Multicast - sending the same message to a defined group of processes.



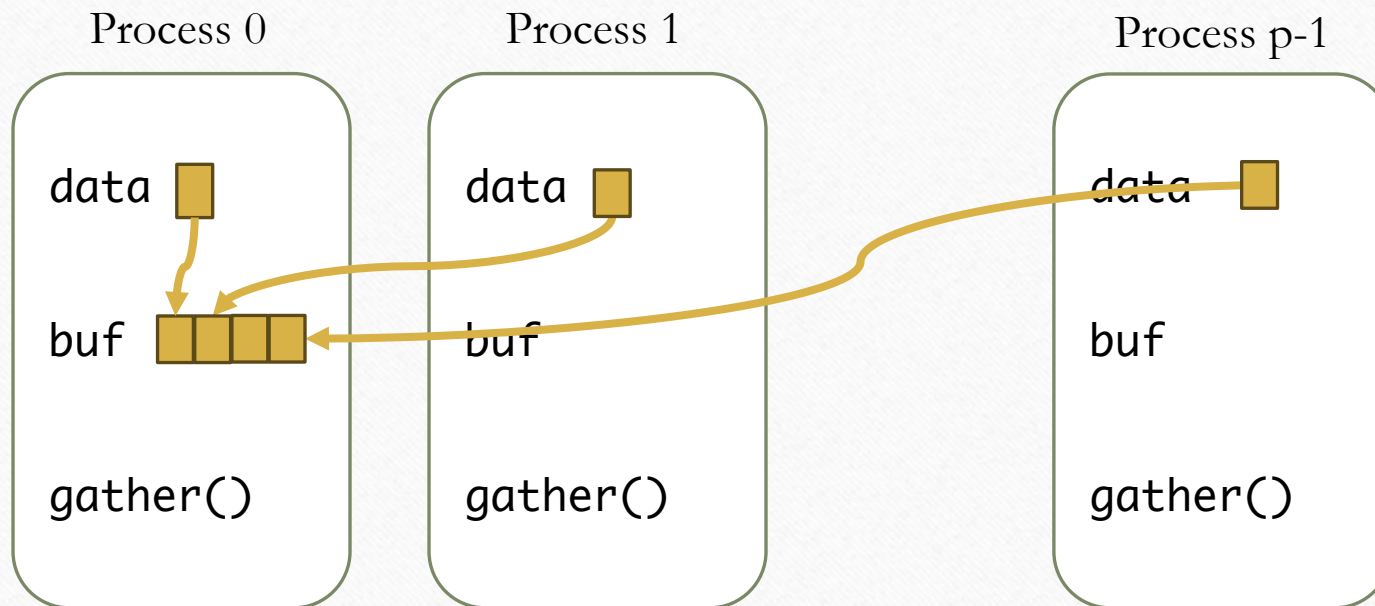
Scatter

- Sending each element of an array in root process to a separate process. Contents of i th location of array sent to i th process.



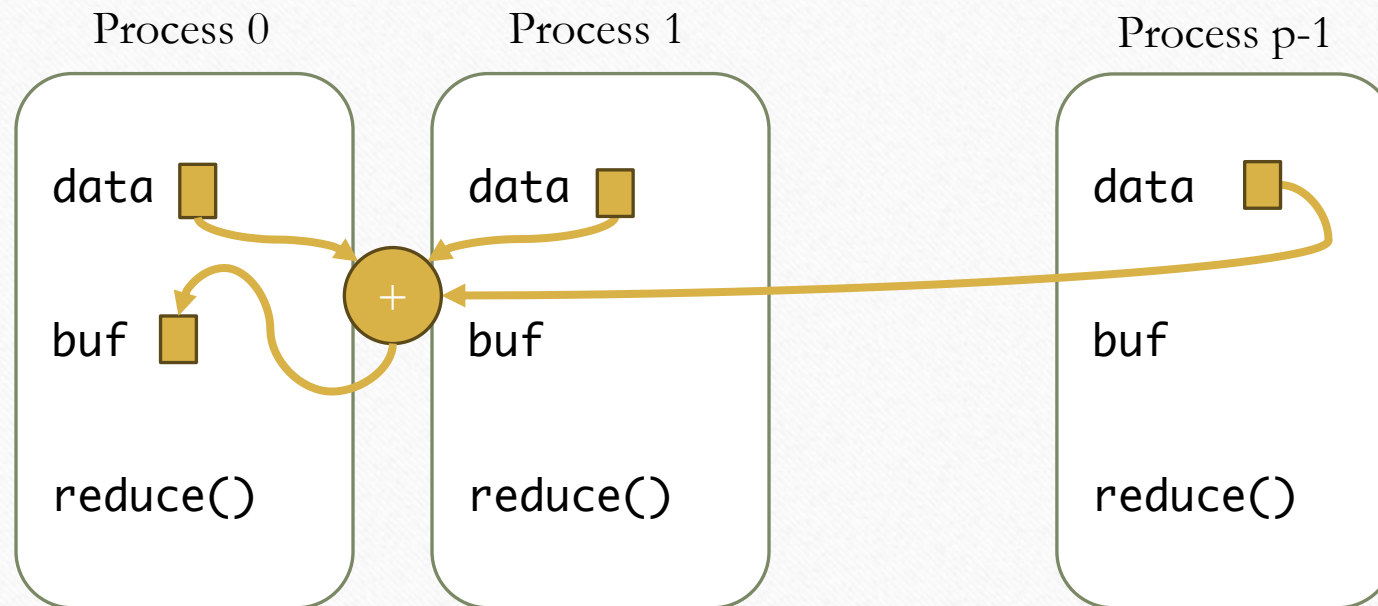
Gather

- Having one process collects individual values from a set of processes.



Reduce

- Gather operation combined with specified arithmetic/logical operation.
- Example: Values could be gathered and then added together by root:



Communicators

- Defines the scope of a communication operation
 - A set of processes that are allowed to communicate between themselves.
- Processes have ranks associated with communication.
- Other communicators can be established for groups of processes.

MPI

From the abstract model to MPI

Communicators

- Defines the scope of a communication operation
 - A set of processes that are allowed to communicate between themselves.
- Processes have ranks associated with communication.
- Other communicators can be established for groups of processes.
- Initially, all processes enrolled in a “universe” called `MPI_COMM_WORLD`, and each process is given a unique rank, a number from 0 to $p - 1$, with p processes.
 - Used in all point-to-point and collective MPI message-passing communications.

Using SPMD Computational Model

```
main (int argc, char *argv[]) {  
    MPI_Init(&argc, &argv);  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank */  
    if (myrank == 0)  
        master();  
    else  
        slave();  
    ...  
    MPI_Finalize();  
}
```

where master() and slave() are to be executed by master and slave processes, respectively.
(see example client-server.cxx)

Signature of blocking MPI_Send (default send)

MPI_Send(buf, count, datatype, dest, tag, comm)

Address of
send buffer

Number of items
to send

Datatype of
each item

Rank of destination
process

Message tag

Communicator

```
paranoid-Macbook-air-M2.local  \%\3
MPI_SEND(3)                      Open MPI
MPI_SEND(3)

MPI_Send - Performs a standard-mode blocking send.

SYNTAX
C Syntax

#include <mpi.h>

int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

Fortran Syntax
:
```

Parameters of blocking receive

MPI_Recv(buf, count, datatype, src, tag, comm, status)

Address of
receive buffer

Maximum number
of items to receive

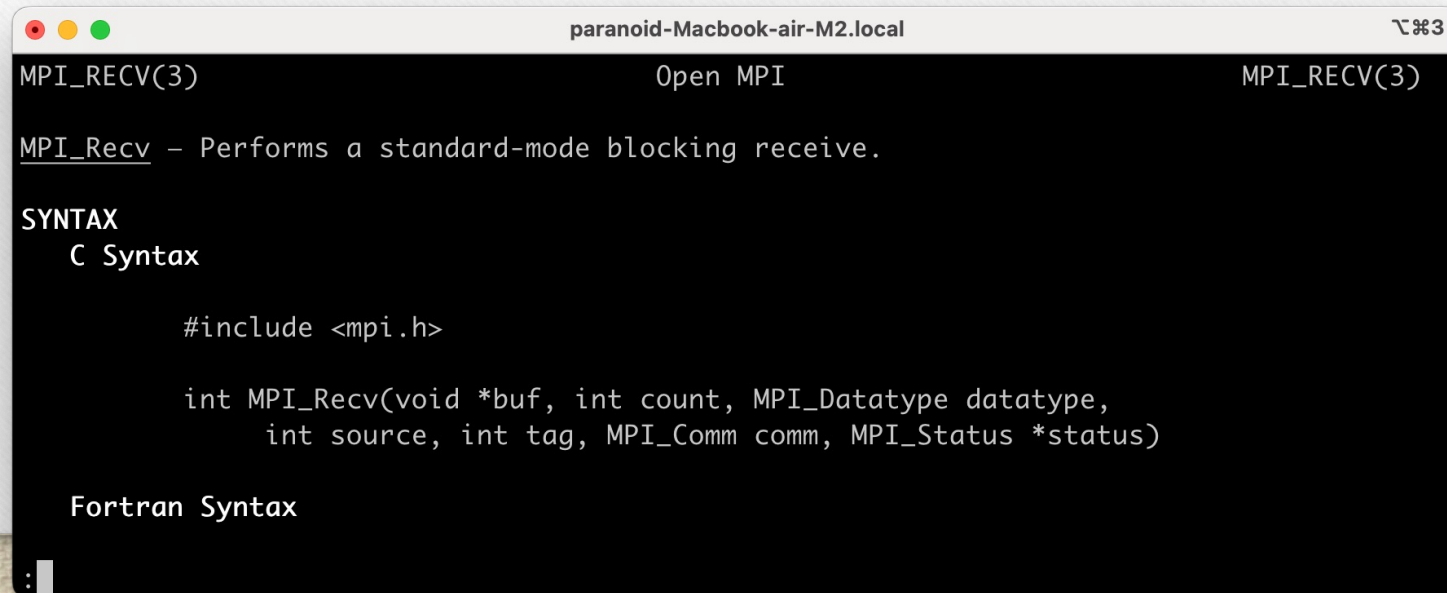
Datatype of
each item

Rank of source
process

Message
tag

Communicator

Status after
operation

A terminal window titled "paranoid-Macbook-air-M2.local" with a window control bar (red, yellow, green buttons) and a zoom icon. The terminal content includes "MPI_RECV(3)" in the title bar, "Open MPI" in the top right, and "MPI_RECV(3)" in the bottom right. The main text describes the MPI_Recv function, its syntax in C and Fortran, and includes a code snippet for the C syntax.

```
MPI_RECV(3)                                Open MPI                                MPI_RECV(3)

MPI_Recv – Performs a standard-mode blocking receive.

SYNTAX
C Syntax

#include <mpi.h>

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)

Fortran Syntax
```


MPI_Status

```
typedef struct _MPI_Status {  
    int count;  
    int cancelled;  
    int MPI_SOURCE;  
    int MPI_TAG;  
    int MPI_ERROR;  
} MPI_Status, *PMPI_Status;
```

Members

count

Number of received entries.

cancelled

Indication if the corresponding request was cancelled.

MPI_SOURCE

Source of the message.

MPI_TAG

Tag value of the message.

MPI_ERROR

Error, associated with the message

Example

To send an integer x from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */

if (myrank == 0) {
    int x;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```


MPI Nonblocking Routines

- **Nonblocking send** - **MPI_Isend()** - will return “immediately” even before source location is safe to be altered.
- **Nonblocking receive** - **MPI_Irecv()** - will return even if no message to accept.

Nonblocking Routine Formats

MPI_Isend(buf, count, datatype, dest, tag, comm, request)

MPI_Irecv(buf, count, datatype, source, tag, comm, request)

Completion detected by **MPI_Wait()** and **MPI_Test()**.

MPI_Wait() waits until operation completed and returns then.

MPI_Test() returns with flag set indicating whether operation completed at that time.

Need to know whether particular operation completed.

Determined by accessing **request** parameter.

Example

To send an integer x from process 0 to process 1 and allow process 0 to continue,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);    /* find rank */
if (myrank == 0) {
    int x;
    MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
    compute(); Compute non deve usare x
    MPI_Wait(req1, status);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```


Send Communication Modes

- **Standard Mode Send** - Not assumed that corresponding receive routine has started. Amount of buffering not defined by MPI. If buffering provided, send could complete before receive reached.
- **Buffered Mode** - Send may start and return before a matching receive. Necessary to specify buffer space via routine `MPI_Buffer_attach()`.
- **Synchronous Mode** - Send and receive can start before each other but can only complete together.
- **Ready Mode** - Send can only start if matching receive already reached, otherwise error. Use with care.

Send Communication Modes

- Each of the four modes can be applied to both blocking and nonblocking send routines.
- Only the standard mode is available for the blocking and nonblocking receive routines.
- Any type of send routine can be used with any type of receive routine.

Collective Communication (MPI-1)

Involves set of processes, defined by an intra-communicator. Message tags not present.
Principal collective operations:

- **MPI_Bcast()** Broadcast from root to all other processes
- **MPI_Gather()** Gather values for group of processes
- **MPI_Scatter()** Scatters buffer in parts to group of processes
- **MPI_Alltoall()** Sends data from all processes to all processes
- **MPI_Reduce()** Combine values on all processes to single value
- **MPI_Reduce_scatter()** Combine values and scatter results
- **MPI_Scan()** Compute prefix reductions of data on processes

Example

To gather items from group of processes into process 0, using dynamically allocated memory in root process:

```
int data[10];           /*data to be gathered from processes*/
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    MPI_Comm_size(MPI_COMM_WORLD, &grp_size); /*find group size*/
    buf = (int *)malloc(grp_size*10*sizeof (int)); /*allocate memory*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_COMM_WORLD) ;
```

MPI_Gather() gathers from all processes, including root.

Evaluating Programs Empirically

Measuring Execution Time

To measure the execution time between point L1 and point L2 in the code, we might have a construction such as

```
L1: time(&t1);    /* start timer */
```

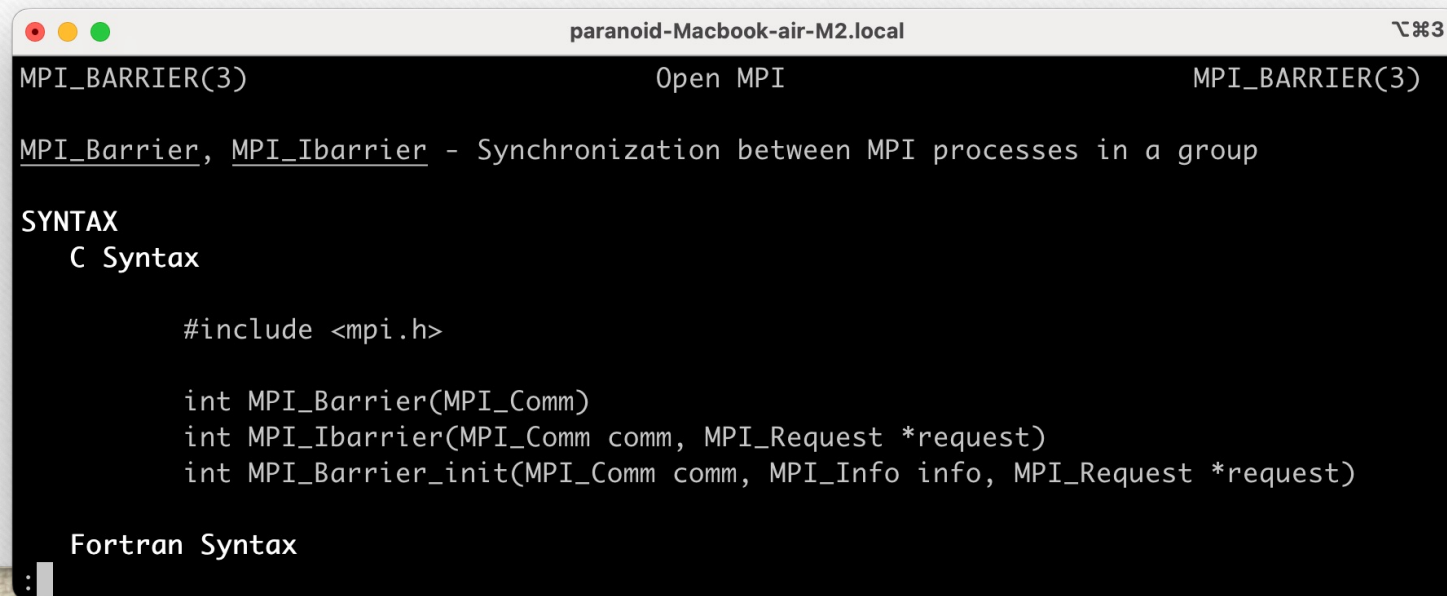
```
L2: time(&t2);    /* stop timer */
```

```
elapsed_time = difftime(t2, t1);    /* elapsed_time = t2 - t1 */  
printf("Elapsed time = %5.2f seconds", elapsed_time);
```

MPI provides the routine **MPI_Wtime()** for returning time (in seconds).

Barrier routine

- A means of synchronizing processes by stopping each one until they all have reached a specific “barrier” call.
- Blocking and non-blocking mode (two-phase - explained later)



```
paranoid-Macbook-air-M2.local  ⌵⌘3
MPI_BARRIER(3)                Open MPI                MPI_BARRIER(3)

MPI_Barrier, MPI_Ibarrier - Synchronization between MPI processes in a group

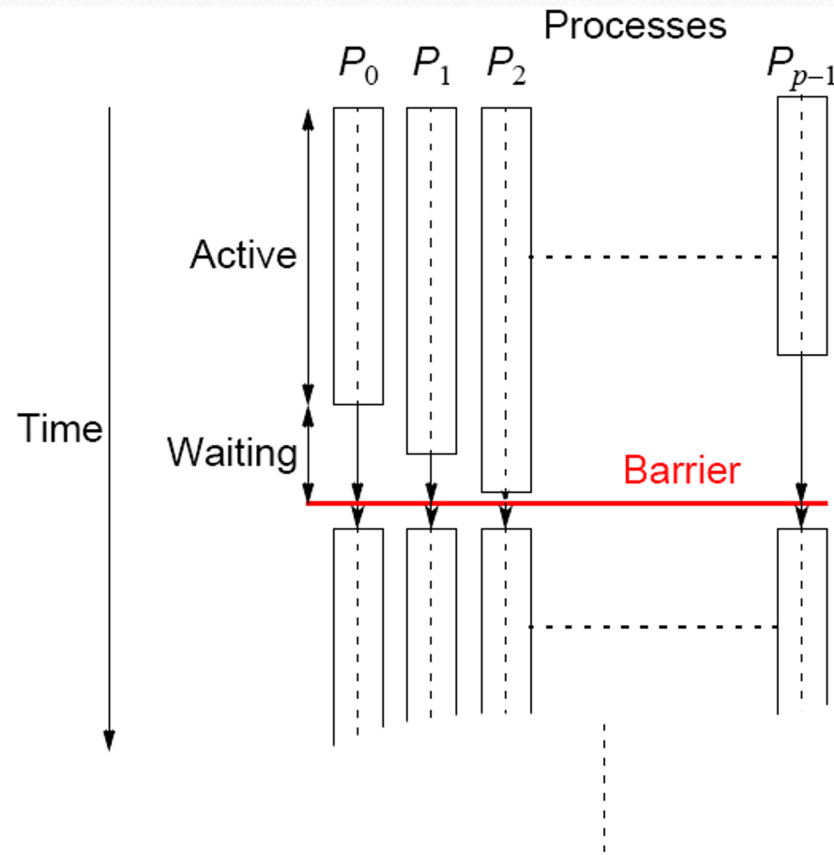
SYNTAX
  C Syntax

    #include <mpi.h>

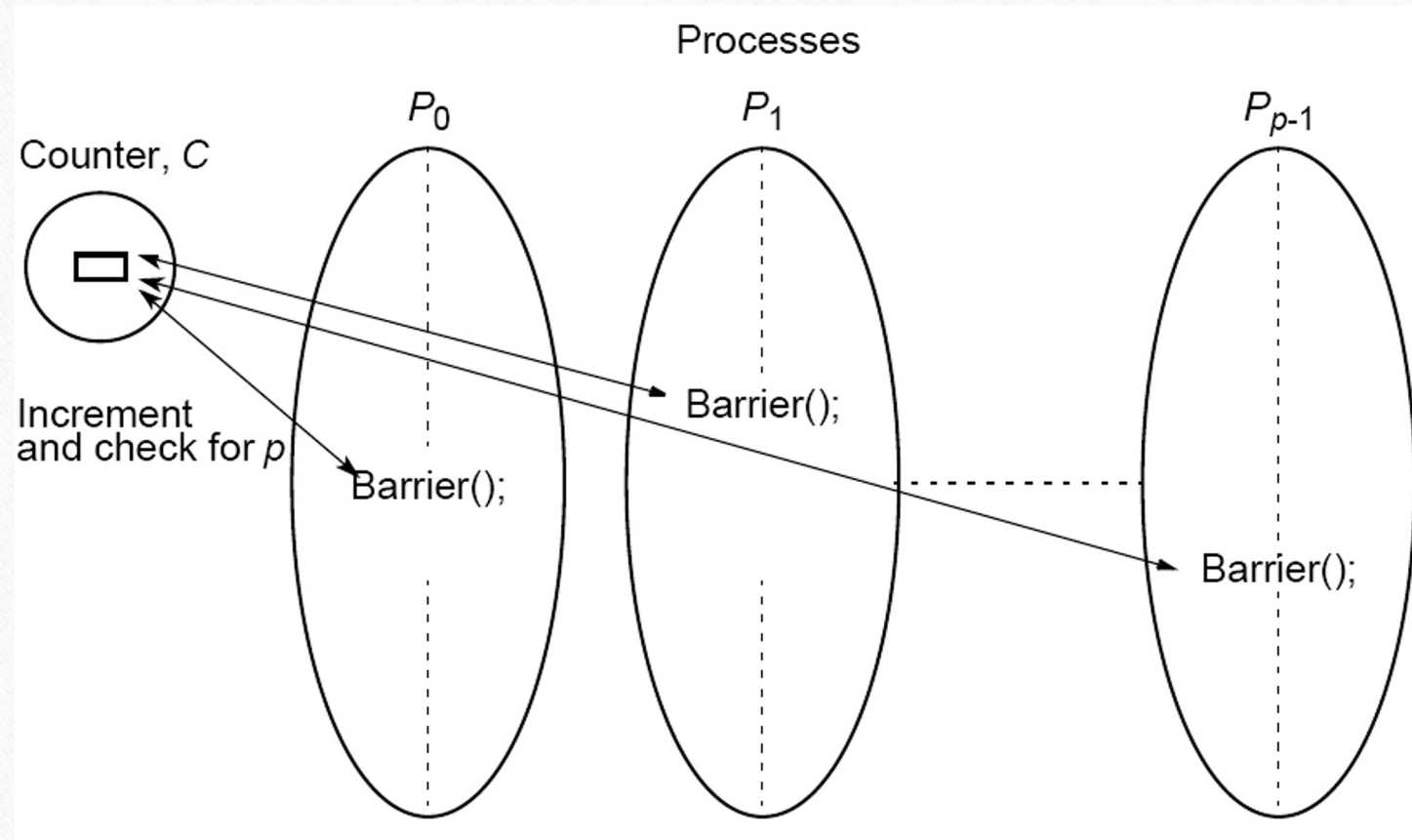
    int MPI_Barrier(MPI_Comm)
    int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
    int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)

  Fortran Syntax
  :
```


Barrier routine



Centralized Implementation (linear)



Barrier: two-phase implementation

- A process enters arrival phase and does not leave this phase until all processes have arrived in this phase.
- Then processes move to departure phase and are released.
- Two-phase handles the reentrant scenario.
 - Good barrier implementations must take into account that a barrier might be used more than once in a process.
 - Might be possible for a process to enter the barrier for a second time before previous processes have left the barrier for the first time.

Barrier: Example implementation

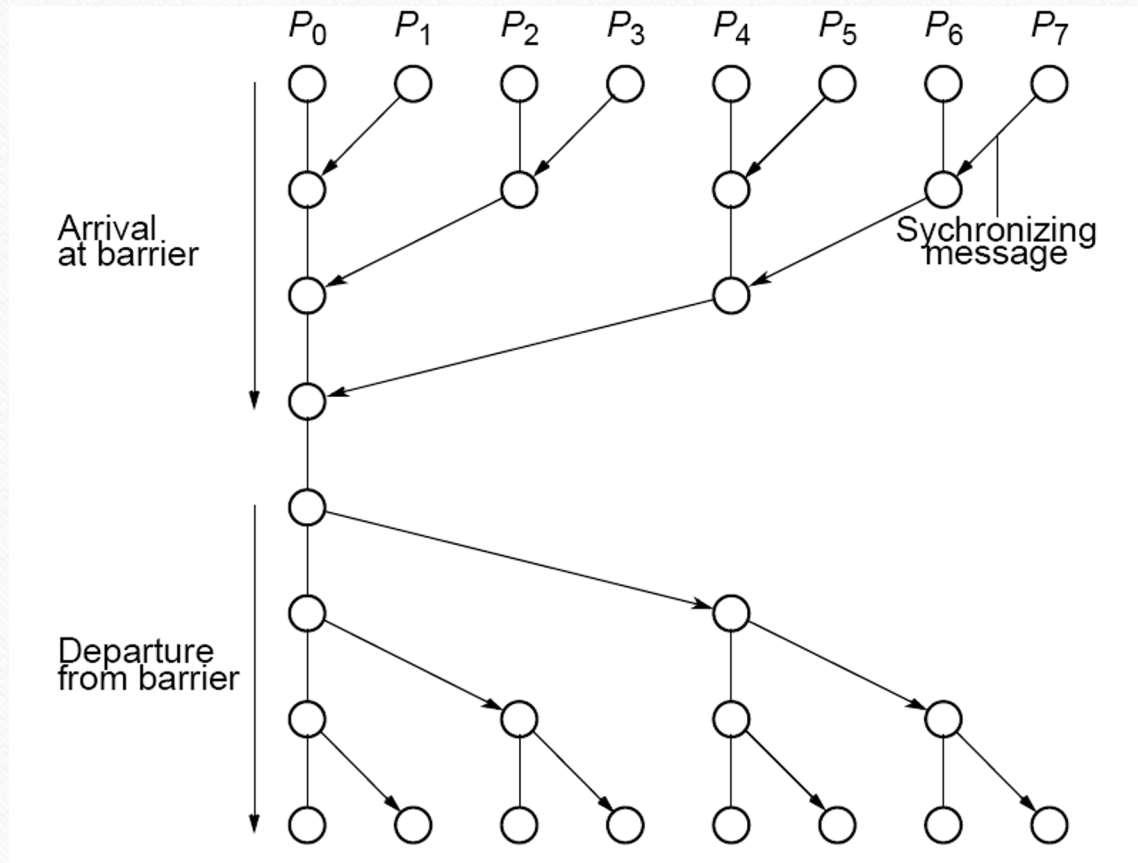
Master

```
for (i = 0; i < n; i++) // count slaves as they reach barrier
    recv(Pany);
for (i = 0; i < n; i++) // release slaves
    send(Pi);
```

Slave processes

```
send(Pmaster);
recv(Pmaster);
```


Barrier: tree implementation



Barrier: Butterfly implementation

1st stage

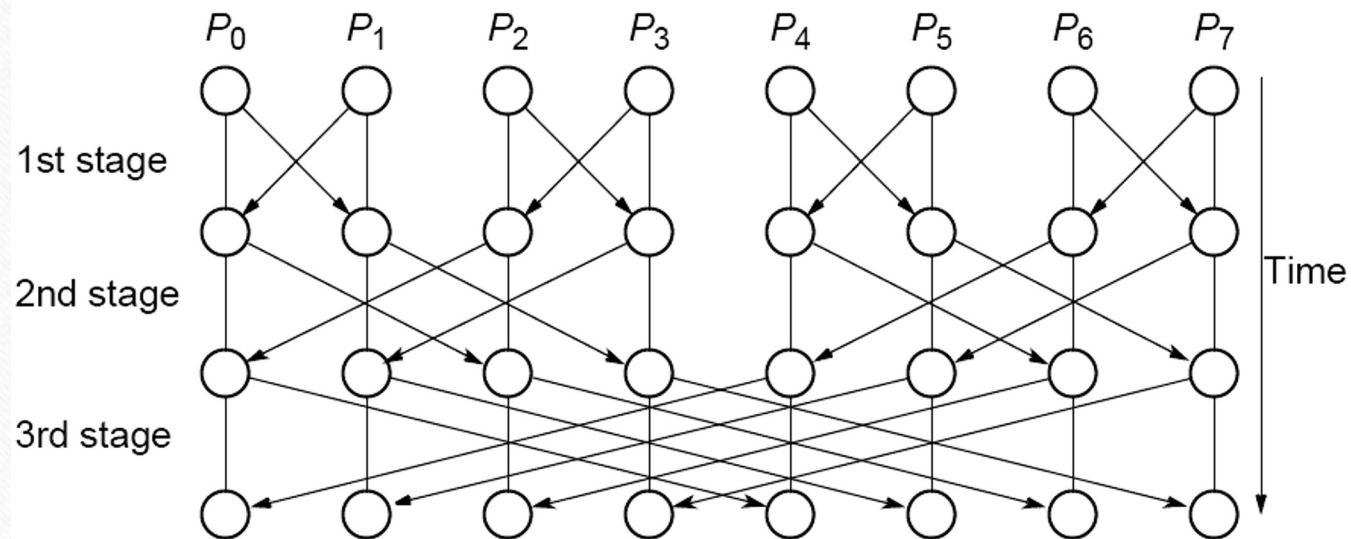
$P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$

2nd stage

$P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$

3rd stage

$P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$



Barrier: tree implementation

More efficient. $O(\log p)$ steps - Suppose 8 processes, $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7$:

1st stage: P_1 sends message to P_0 ; (when P_1 reaches its barrier)

P_3 sends message to P_2 ; (when P_3 reaches its barrier)

P_5 sends message to P_4 ; (when P_5 reaches its barrier)

P_7 sends message to P_6 ; (when P_7 reaches its barrier)

2nd stage: P_2 sends message to P_0 ; (P_2 & P_3 reached their barrier)

P_6 sends message to P_4 ; (P_6 & P_7 reached their barrier)

3rd stage: P_4 sends message to P_0 ; (P_4, P_5, P_6 , & P_7 reached barrier)

P_0 terminates arrival phase;

(when P_0 reaches barrier & received message from P_4)

Release with a reverse tree construction.

MPI-2: New features

- ***Dynamic Processes*** - extensions that remove the static process model of MPI. Provides routines to create new processes after job startup.
- ***One-Sided Communications*** - provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations.
- ***Extended Collective Operations*** - allows for the application of collective operations to inter-communicators
- ***External Interfaces*** - defines routines that allow developers to layer on top of MPI, such as for debuggers and profilers.
- ***Additional Language Bindings*** - describes C++ bindings and discusses Fortran-90 issues.
- ***Parallel I/O*** - describes MPI support for parallel I/O.

MPI-3: New features

- *Nonblocking Collective Operations* - permits tasks in a collective to perform operations without blocking, possibly offering performance improvements.
- *New One-sided Communication Operations* - to better handle different memory models.
- *Neighborhood Collectives* - extends the distributed graph and Cartesian process topologies with additional communication power.
- *Fortran 2008 Bindings* - expanded from Fortran90 bindings
- *MPIT Tool Interface* - allows the MPI implementation to expose certain internal variables, counters, and other states to the user (most likely performance tools).
- *Matched Probe* - fixes an old bug in MPI-2 where one could not probe for messages in a multi-threaded environment.

Compiling/executing (SPMD) MPI program

To run a MPI executable a launcher is needed:

- Interactive: `mpirun`
- Batch: PMIX (apply to several workload managers, including SLURM)_x

To compile MPI programs:

```
mpicc -o file file.c
```

```
mpicxx -o file file.cpp
```

To execute MPI program:

```
mpirun -v -np no_processors file
```


Interactive shell (mpirun)

- Before starting MPI for the first time, need to create a hostfile describing the target hosts. Syntax depends on MPI implementation. Here we assume OpenMPI
- The number of slots can be determined by the number of cores on the node or the number of processor sockets. If no slots are specified for a host, then the number of slots defaults to one. In this example, a host list file called myhosts specifies three nodes, and each node has two slots:

```
cat myhosts  
node1 slots=2  
node2 slots=2  
node3 slots=2
```

```
mpirun -np 6 -hostfile myhost ./a.out
```

Batch (SLURM)

- See slidedeck on SLURM