Message-passing

Message-passing multiprocessors

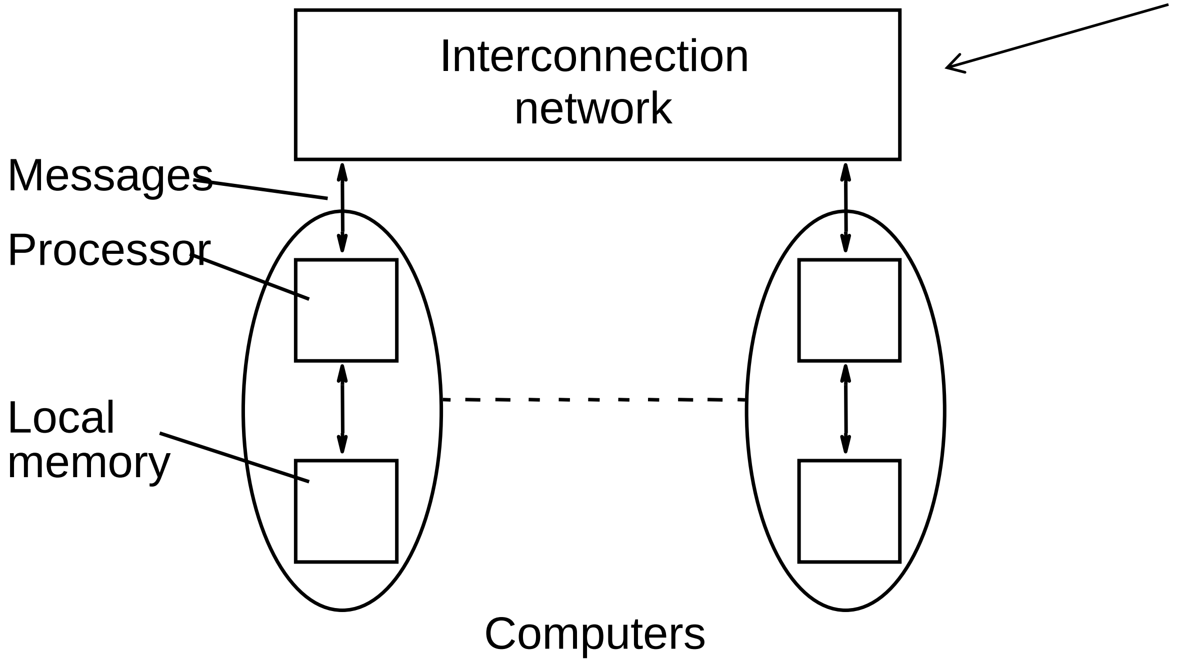
Message-passing computing and MPI

Adapted from slides by Barry Wilkinson, University of North Carolina at Charlotte, author of PARALLEL PROGRAMMING TECHNIQUES AND APPLICATIONS USING NETWORKED WORKSTATIONS AND PARALLEL COMPUTERS 2nd Edition, Pearson, 2005

Message-Passing Multicomputer

Message-Passing Multicomputer

Complete computers connected through an interconnection network:



Many interconnection networks explored in the 1970s and 1980s including 2- and 3-dimensional meshes, hypercubes, and multistage interconnection networks

Networked Computers as a Computing Platform

- A network of computers became a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing in early 1990s.

- Several early projects. Notable:

  - Berkeley NOW (network of workstations) project.
  - NASA Beowulf project.

Key advantages:

- Very high performance workstations and PCs readily available at low cost.

- The latest processors can easily be incorporated into the system as they become available.

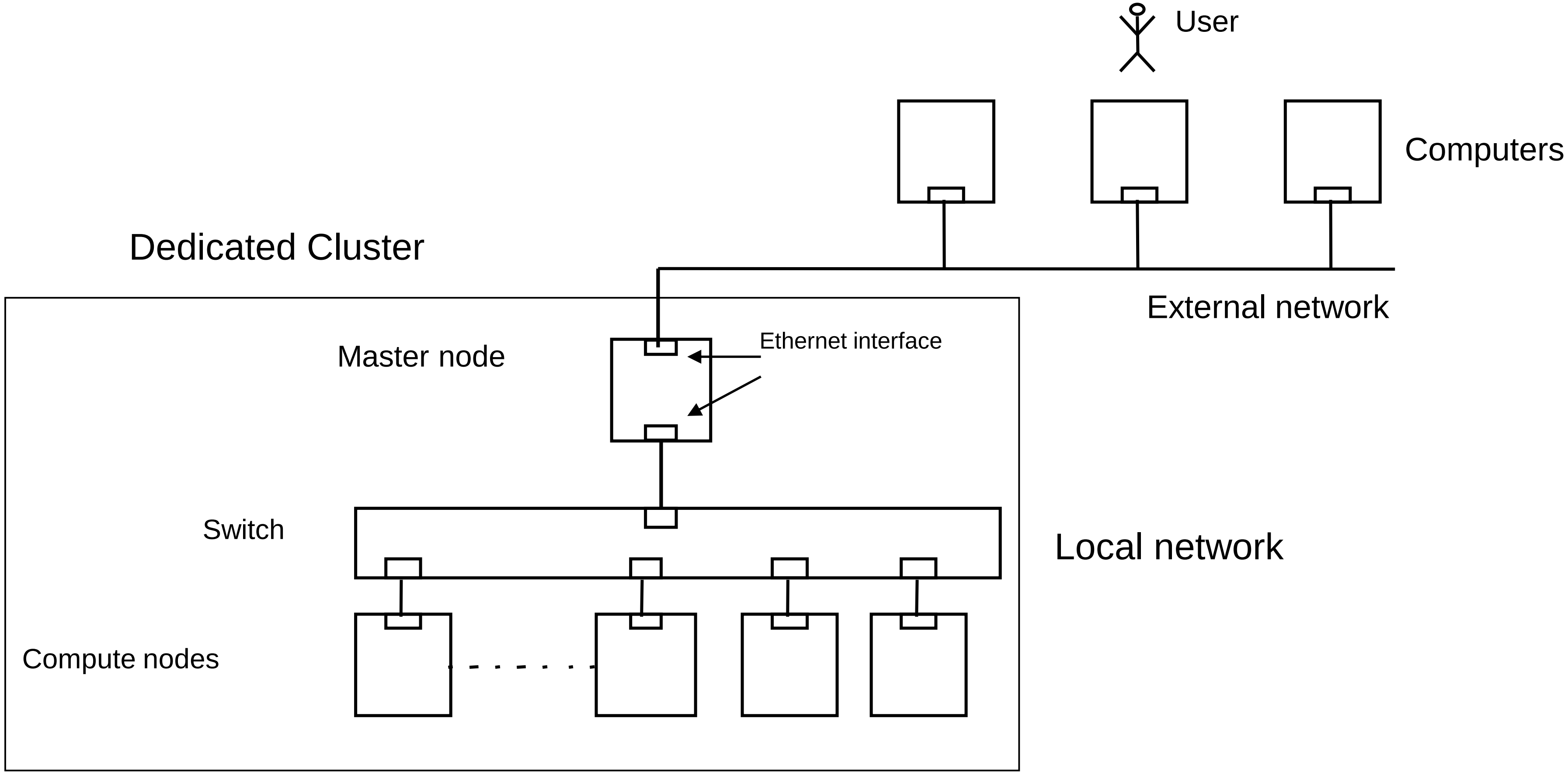
- Existing software can be used or modified.

Cluster Interconnects

- Originally fast Ethernet on low cost clusters
- Gigabit Ethernet - easy upgrade path

More specialized/higher performance interconnects available including Myrinet and Infiniband.

Dedicated cluster with a master node and compute nodes



Software Tools for Clusters

- Based upon message passing programming model
- User-level libraries provided for explicitly specifying messages to be sent between executing processes on each computer .
- Use with regular programming languages (C, C++, ...).
- Can be quite difficult to program correctly as we shall see.

Message-Passing Computing

Software Tools for Clusters

Late 1980's	Parallel Virtual Machine (PVM) - developed Became very popular.
Mid 1990's -	Message-Passing Interface (MPI) - standard defined.

Based upon Message Passing Parallel Programming model.

Both provide a set of user-level libraries for message passing. Use with sequential programming languages (C, C++, ...).

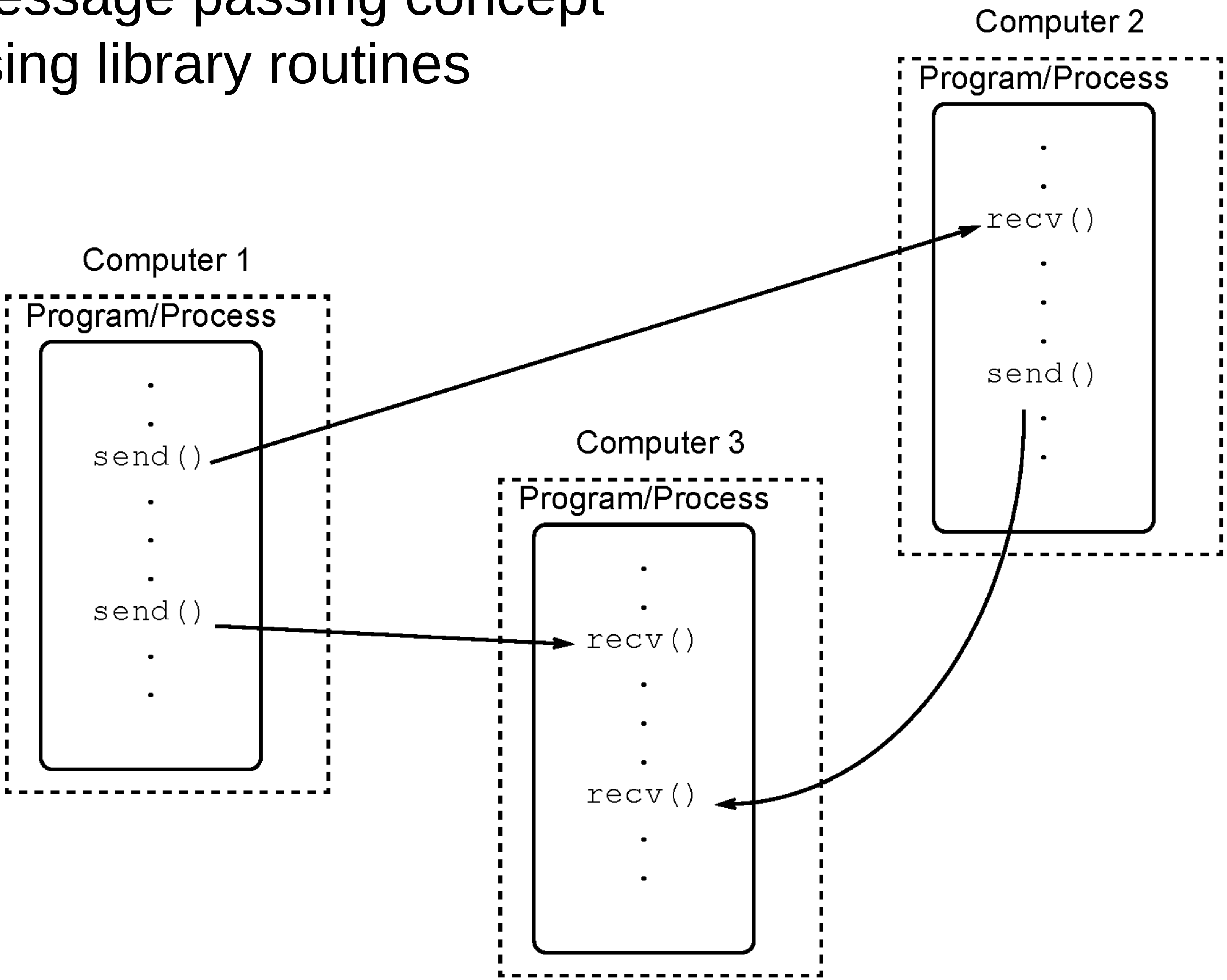
MPI
(Message Passing Interface)

- Message passing library standard developed by group of academics and industrial partners to foster more widespread use and portability.

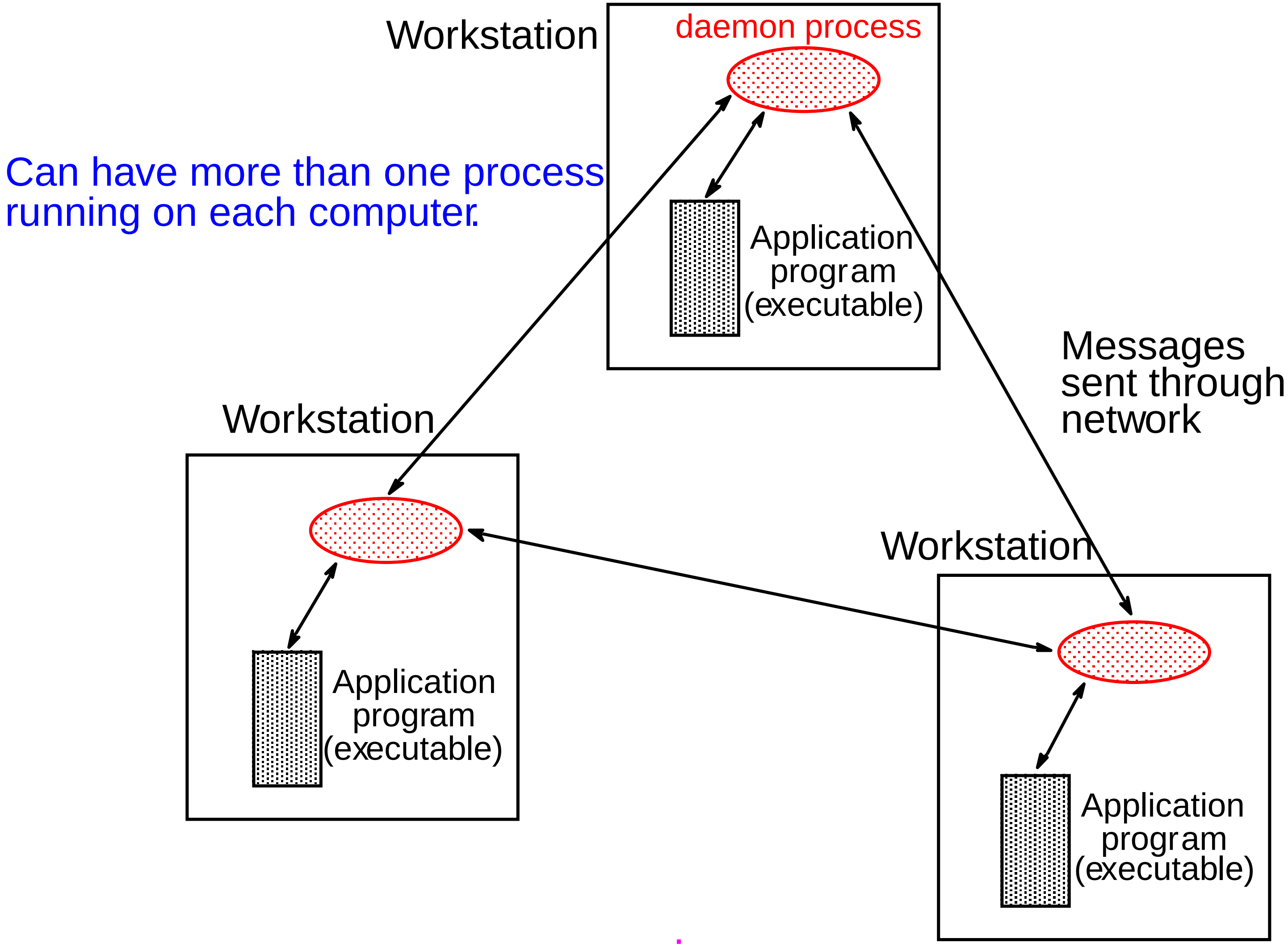
- Defines routines, not implementation.

- Several free implementations exist.

Message passing concept using library routines



Message routing between computers typically done by daemon processes installed on computers that form the "virtual machine".



Message-Passing Programming using User-level Message-Passing Libraries

Two primary mechanisms needed:

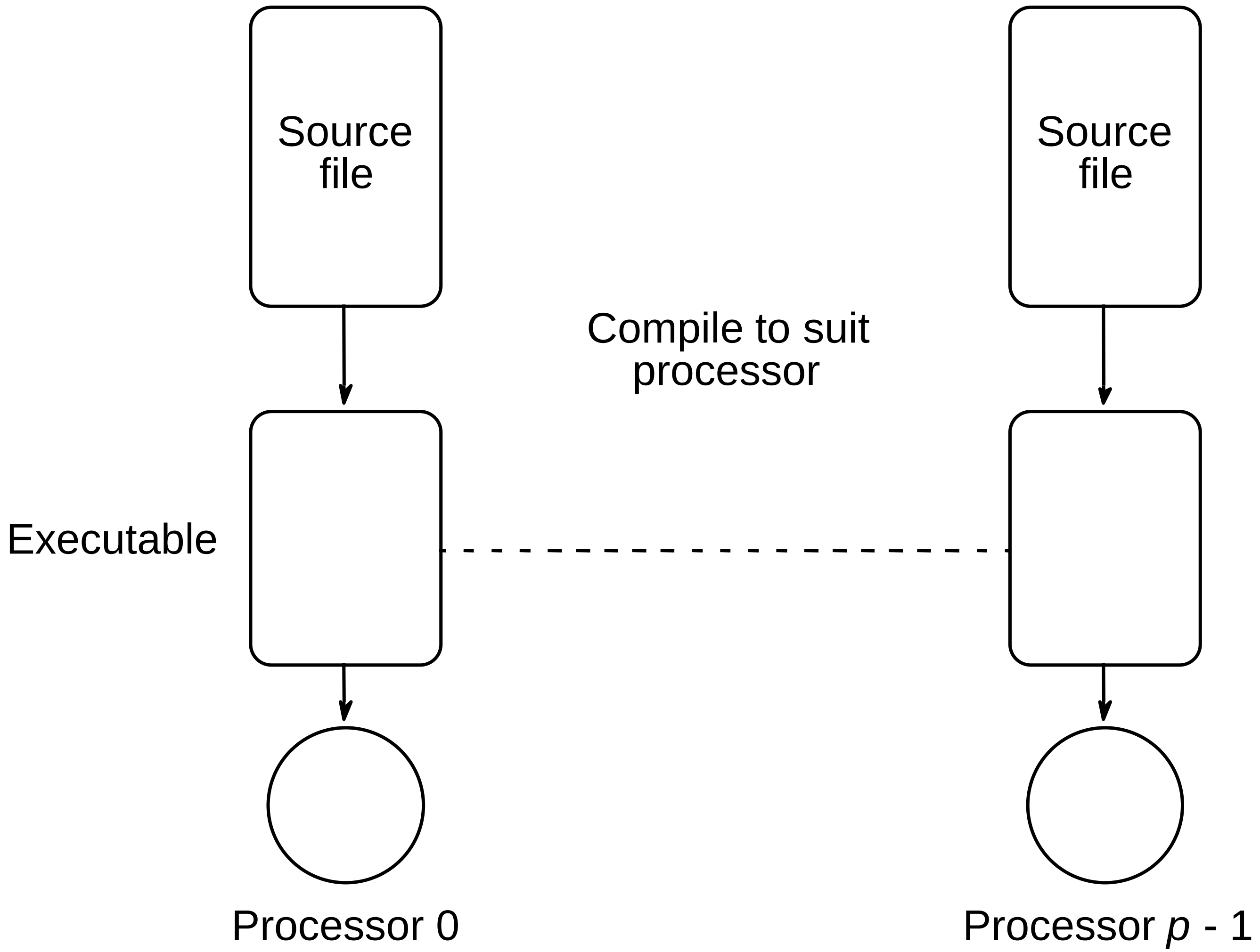
1. A method of creating processes for execution on different computers

2. A method of sending and receiving messages

Creating processes on different computers

Multiple program, multiple data (MPMD) model

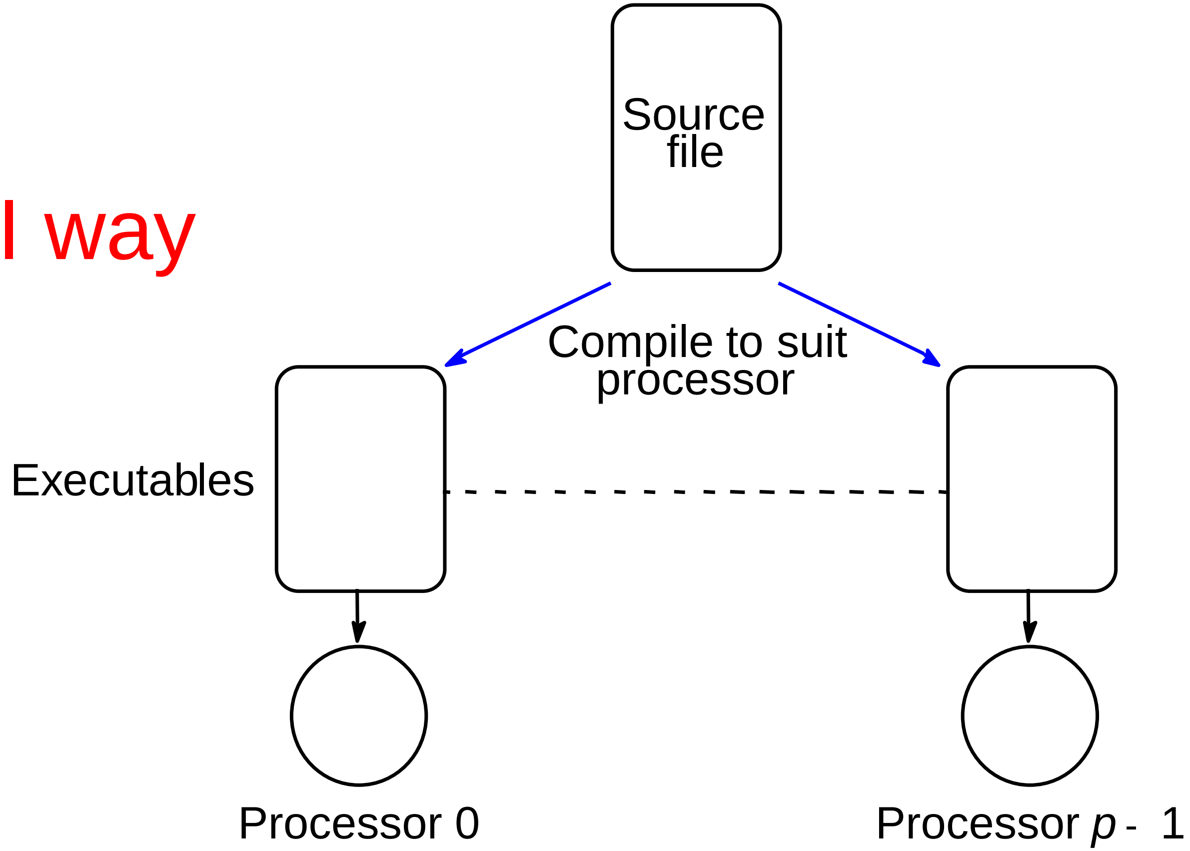
- Different programs executed by each processor



Single Program Multiple Data (SPMD) model

- Same program executed by each processor
- Control statements select different parts for each processor to execute.

Basic MPI way



In MPI, processes within a defined communicating group given a number called a rank starting from zero onwards.

Program uses control constructs, typically IF statements, to direct processes to perform specific actions.

Example

```
if (rank == 0) ...          /* do this */;
if (rank == 1) ...          /* do this */;
       .
       .
       .
```

Master-Slave approach

Usually computation constructed as a master-slave model

One process (the master), performs one set of actions and all the other processes (the slaves) perform identical actions although on different data, i.e.

```
if (rank == 0) ...      /* master do this */;
else ...                /* all slaves do this */;
```

Static process creation

- All executables started together.
- Done when one starts the compiled programs.
- Normal MPI way.

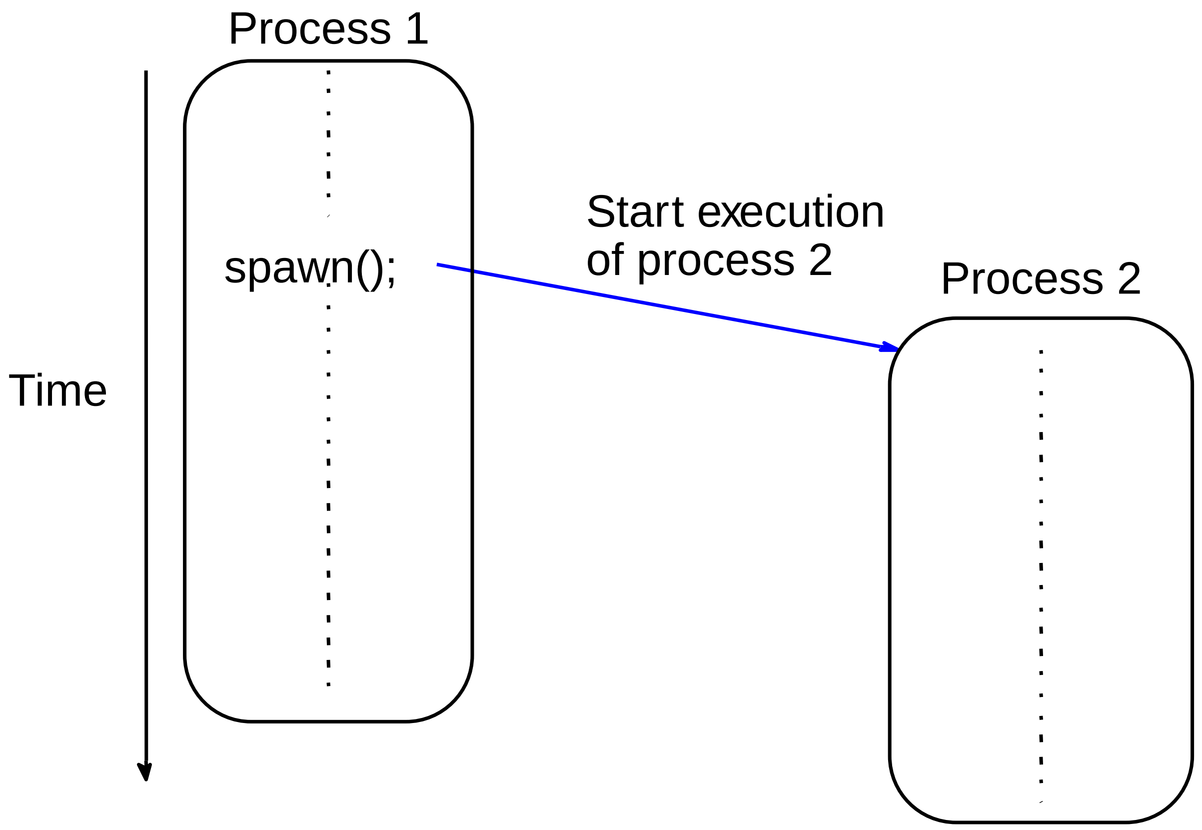
Multiple Program Multiple Data (MPMD) Model with Dynamic Process Creation

- One processor executes master process.
- Other processes started from within master process

Available in MPI-2

Might find applicability if do not initially how many processes needed.

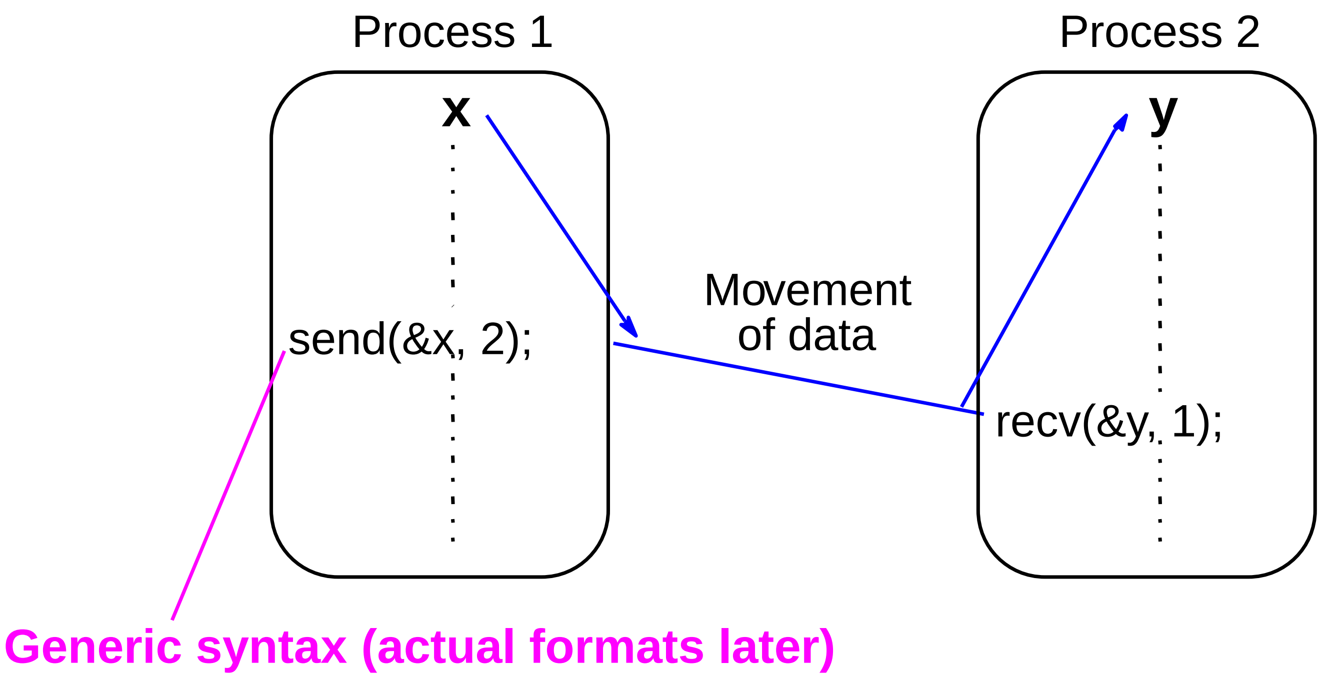
Does have a process creation overhead.



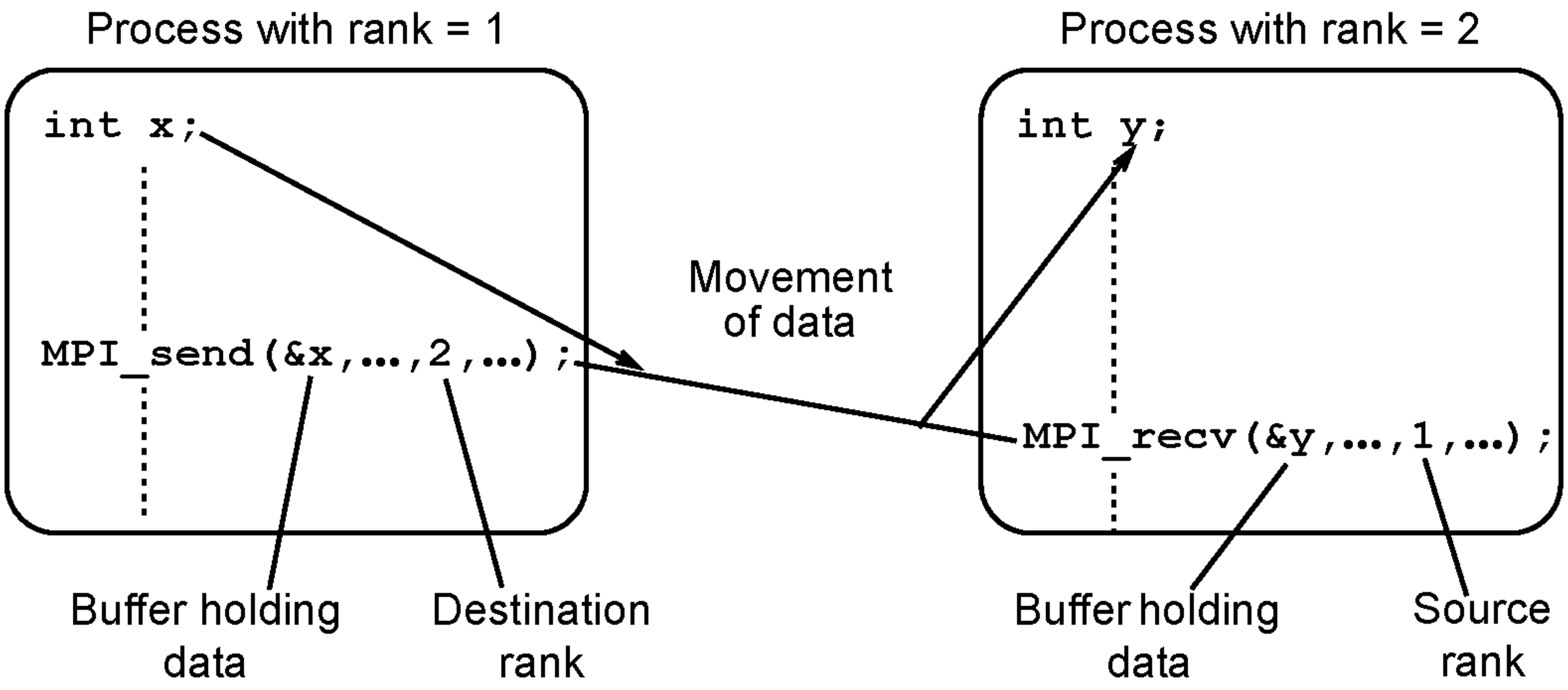
Methods of sending and receiving messages

Basic "point-to-point"
Send and Receive Routines

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



MPI point-to-point message passing using MPI_send() and MPI_recv() library calls



Semantics of MPI_Send() and MPI_Recv()

Called blocking, which means in MPI that routine waits until all its local actions have taken place before returning.

After returning, any local variables used can be altered without affecting message transfer.

MPI_Send() - Message may not reached its destination but process can continue in the knowledge that message safely on its way.

MPI_Recv() – Returns when message received and data collected. Will cause process to stall until message received.

Other versions of MPI_Send() and MPI_Recv() have different semantics.

Message Tag

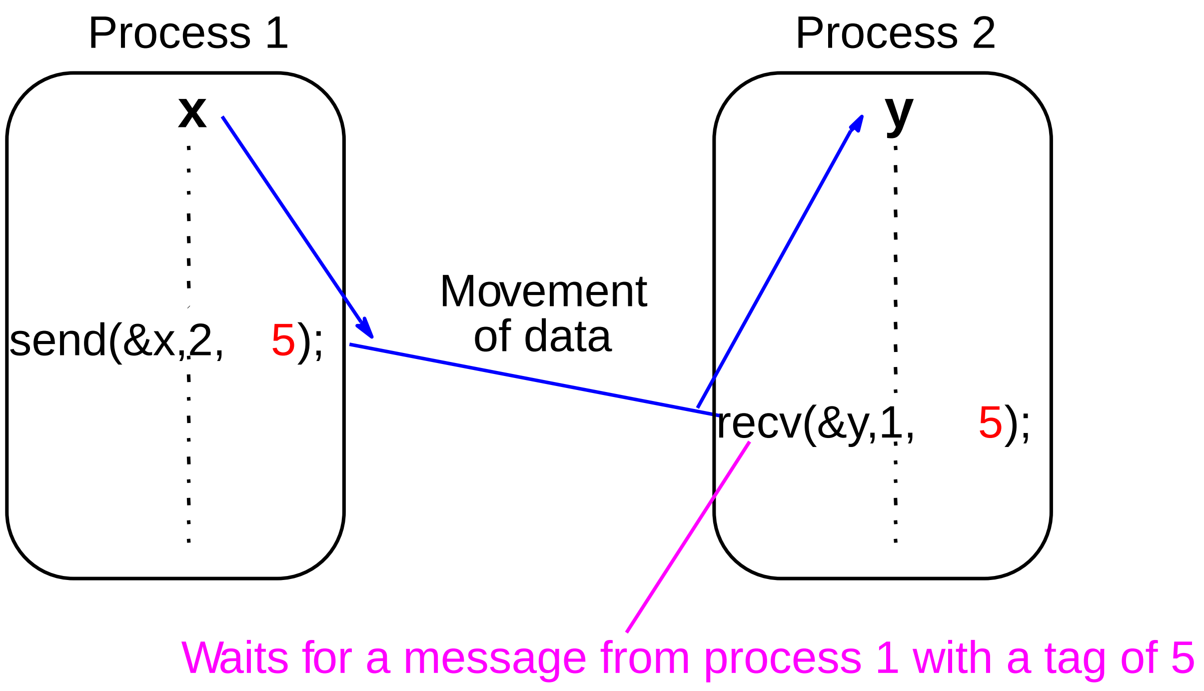
- Used to differentiate between different types of messages being sent.

- Message tag is carried within message.

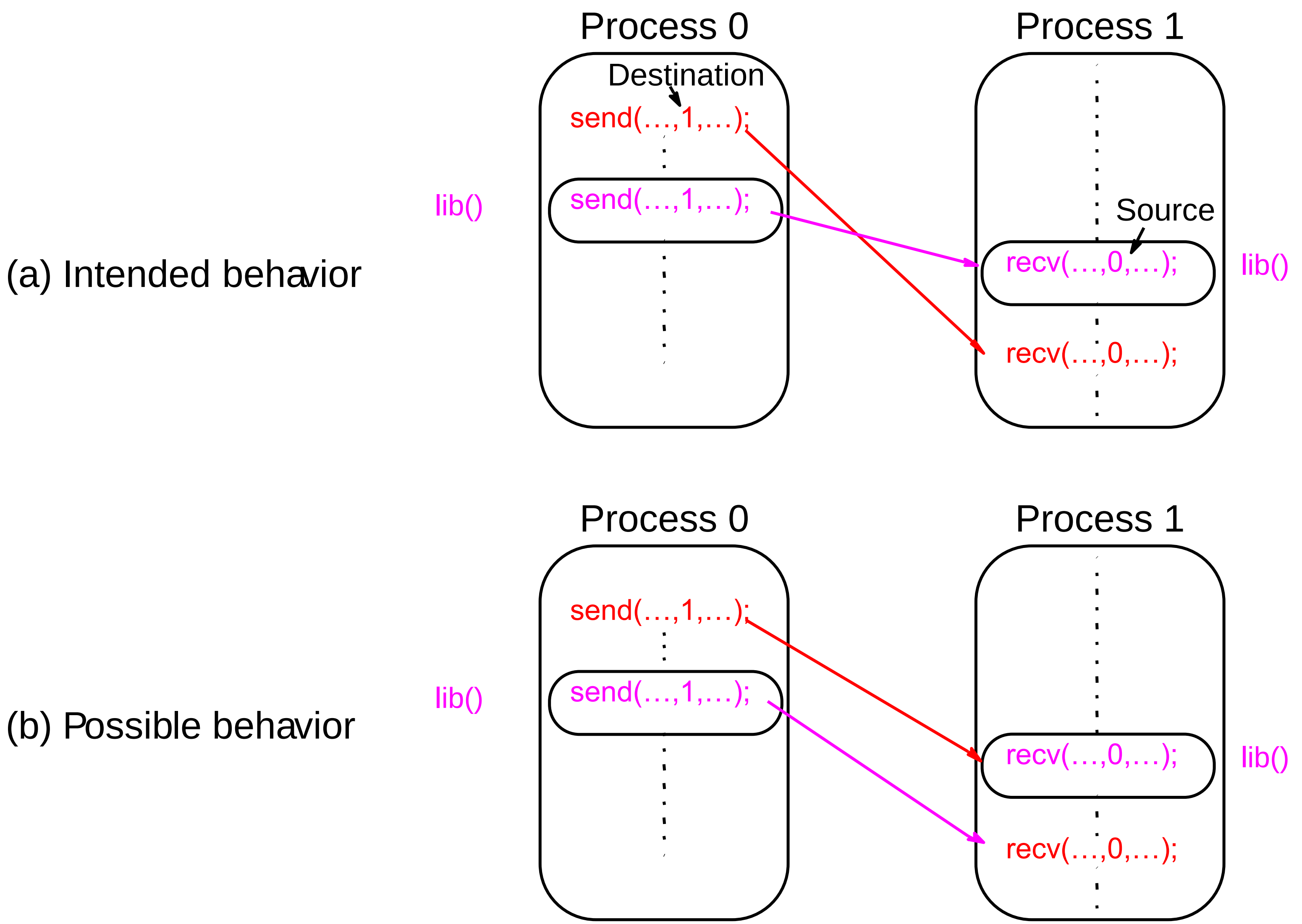
- If special type matching is not required, a wild card message tag used. Then recv() will match with any send().

Message Tag Example

To send a message, x, with message tag 5 from a source process, 1, to a destination process, 2, and assign to y:



Unsafe message passing - Example



MPI Solution
“Communicators”

- Defines a communication domain - a set of processes that are allowed to communicate between themselves.

- Communication domains of libraries can be separated from that of a user program.

- Used in all point-to-point and collective MPI message-passing communications.

Note: **Intracommunicator** – for communicating within a single group of processes.
Intercommunicator - for communicating within two or more groups of processes

Default Communicator MPI_COMM_WORLD

- Exists as first communicator for all processes existing in the application.
- A set of MPI routines exists for forming communicators.
- Processes have a “rank” in a communicator.

Using SPMD Computational Model

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

where master() and slave() are to be executed by master process and slave process, respectively.

Parameters of blocking 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

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

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);
}
```

Sample MPI Hello World program

```
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv ) {
    char message[20];
    int i,rank, size, type=99;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank == 0) {
        strcpy(message, "Hello, world");
        for (i=1; i<size; i++)
            MPI_Send(message,13,MPI_CHAR,i,type,MPI_COMM_WORLD);
    } else
        MPI_Recv(message,20,MPI_CHAR,0,type,MPI_COMM_WORLD,&status);
    printf( "Message from process =%d : %.13s\n", rank,message);
    MPI_Finalize();
}
```

Program sends message “Hello World” from master process (rank = 0) to each of the other processes (rank != 0). Then, all processes execute a println statement.

In MPI, standard output automatically redirected from remote computers to the user’s console so final result will be

```
Message from process =1 : Hello, world
Message from process =0 : Hello, world
Message from process =2 : Hello, world
Message from process =3 : Hello, world
...
```

except that the order of messages might be different but is unlikely to be in ascending order of process ID; it will depend upon how the processes are scheduled.

Setting Up the Message Passing Environment

Usually computers specified in a file, called a hostfile or machines file.

File contains names of computers and possibly number of processes that should run on each computer.

Implementation-specific algorithm selects computers from list to run user programs.

Users may create their own machines file for their program.

Example

coit-grid01.uncc.edu

coit-grid02.uncc.edu

coit-grid03.uncc.edu

coit-grid04.uncc.edu

coit-grid05.uncc.edu

If a machines file not specified, a default machines file used or it may be that program will only run on a single computer.

Compiling/Executing MPI Programs

- Minor differences in the command lines required depending upon MPI implementation.
- For the assignments, we will use OpenMPI.
- Generally, a machines file need to be present that lists all the computers to be used. MPI then uses those computers listed. Otherwise it will simply run on one computer

OpenMPI Commands

Two basic commands:

- mpicc, a script to compile MPI programs
- mpiexec - MPI-2 standard command *

* mpiexec replaces earlier mpirun comamnd although mpirun still exists.)

Compiling/executing (SPMD) MPI program

To start MPI: Nothing special.

(Make sure mpd daemons running)

To compile MPI programs:

for C `mpicc -o prog prog.c`

for C++ `mpiCC -o prog prog.cpp`

To execute MPI program:

A positive integer

```
mpiexec -n no_procs prog
```

mpiexec -machinefile machines -n 4 prog

would run prog with four processes.

Each processes would execute on one of machines in list.

MPI would cycle through list of machines giving processes to machines.

Can also specify number of processes on a particular machine by adding that number after machine name.)

Debugging/Evaluating Parallel Programs Empirically

Evaluating Programs Empirically
Measuring Execution Time

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

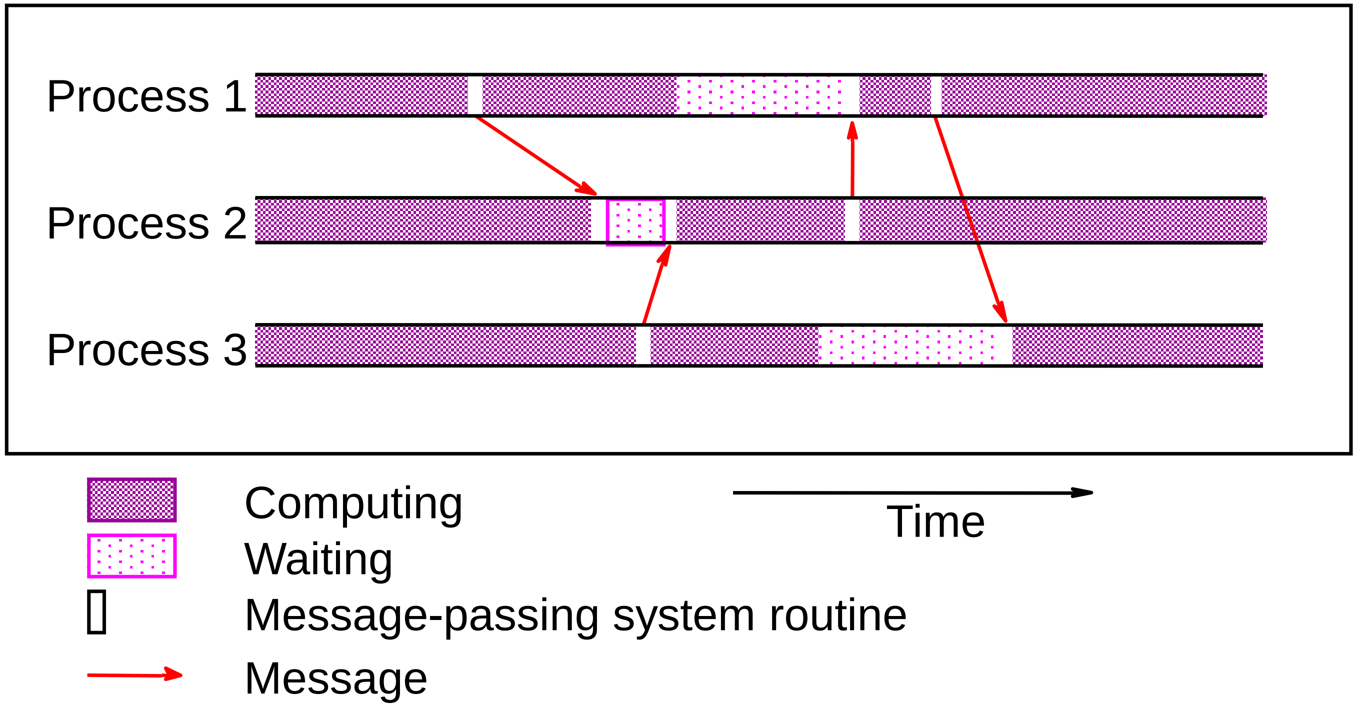
```
L1: time(&t1);          /* start timer */
      .
      .
L2: time(&t2);        /* stop timer */
          .
elapsed_Time = difftime(t2, t1); /*time=t2-t1*/

printf("Elapsed time=%5.2f secs",elapsed_Time);
```

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

```
double start_time, end_time, exe_time;

start_time = MPI_Wtime();

    .
    .

end_time = MPI_Wtime();
exe_time = end_time - start_time;
```

Visualization Tools

Programs can be watched as they are executed in a space-time diagram (or process-time diagram):



Visualization tools available for MPI, e.g., Upshot.

Message-Passing Computing

More MPI routines:

Collective routines
Synchronous routines
Non-blocking routines

Collective message-passing routines

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 routines not absolutely necessary.

Collective Communication

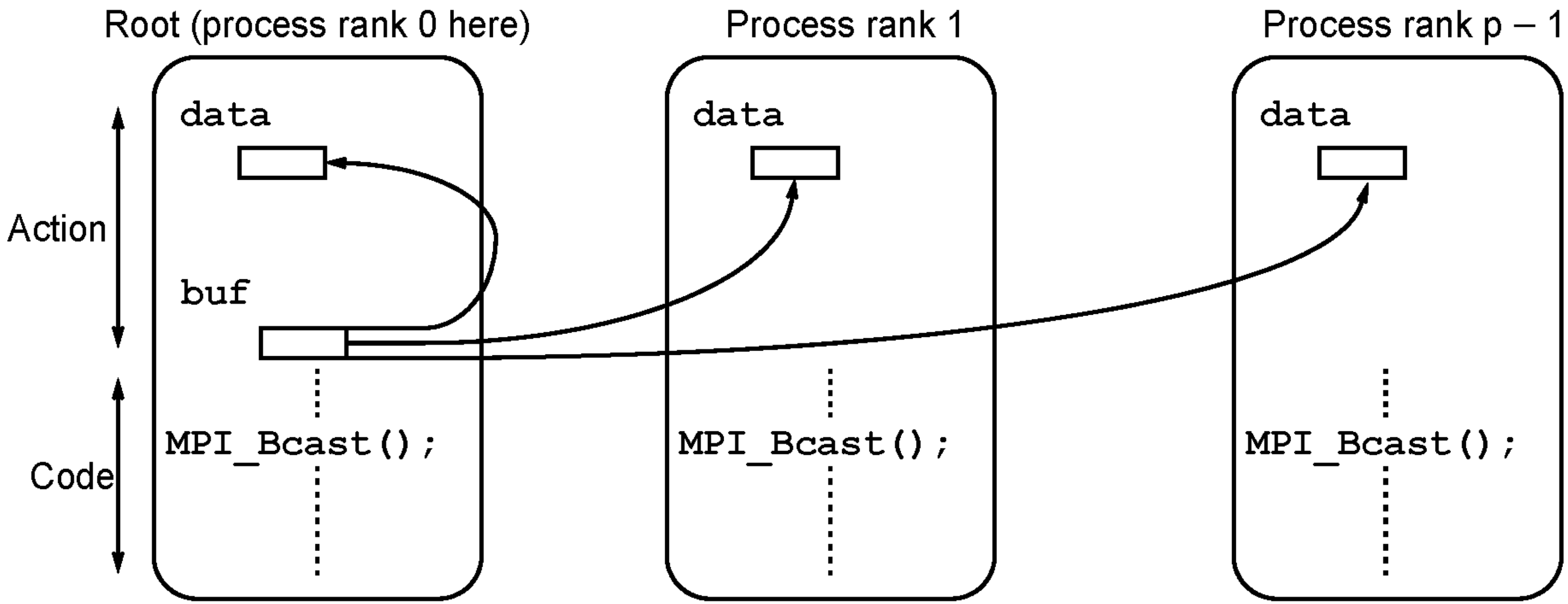
Involves set of processes, defined by an intra-communicator. Message tags not present. Principal collective operations:

- **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

- **MPI_Barrier()** - A means of synchronizing processes by stopping each one until they all have reached a specific “barrier” call.

MPI broadcast operation

Sending same message to all processes in communicator.
Multicast - sending same message to defined group of processes.



MPI_Bcast parameters

```
MPI_Bcast(*buf, count, datatype, root, comm)
```

Address of send buffer

Number of items to send

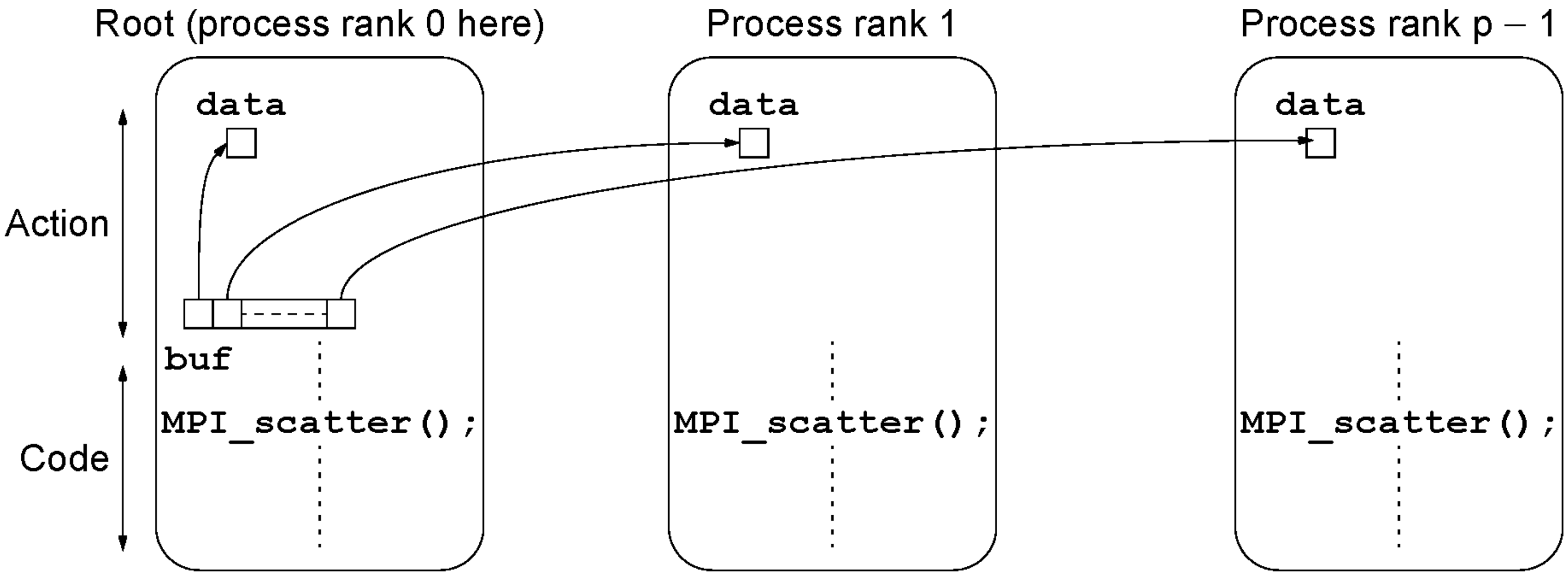
Datatype of each item

Rank root process (source of broadcast)

Communicator

Basic MPI scatter operation

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.



MPI scatter parameters

```
MPI_Scatter(*sbuf,scount,stype,*rbuf,rcount,rtype,root,comm)
```

Address of send buffer

Number of items to send to each process

Datatype of items sent

Address of receive buffer

Number of items to receive

Datatype of items received

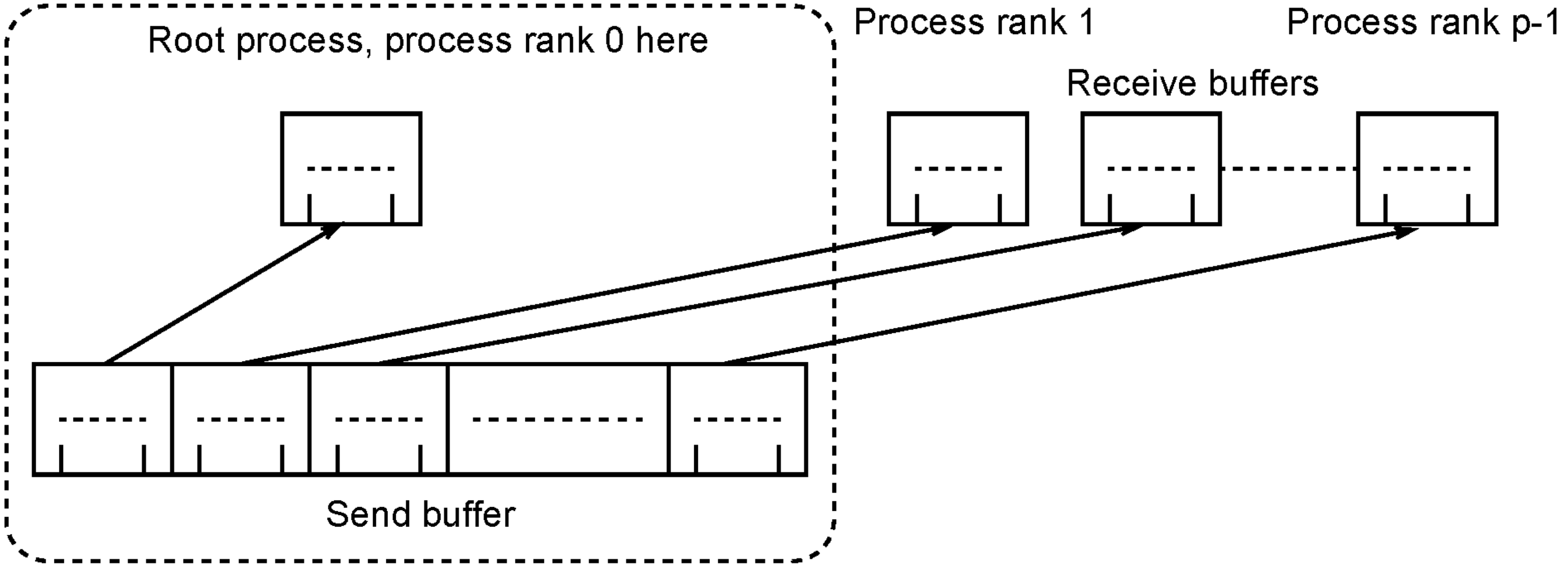
Rank of root process (source)

Communicator

- Simplest scatter would be as illustrated, which one element of an array is sent to different processes.

- Extension in MPI_Scatter() routine is to send a fixed number of contiguous elements to each process.

Scattering contiguous groups of elements to each process



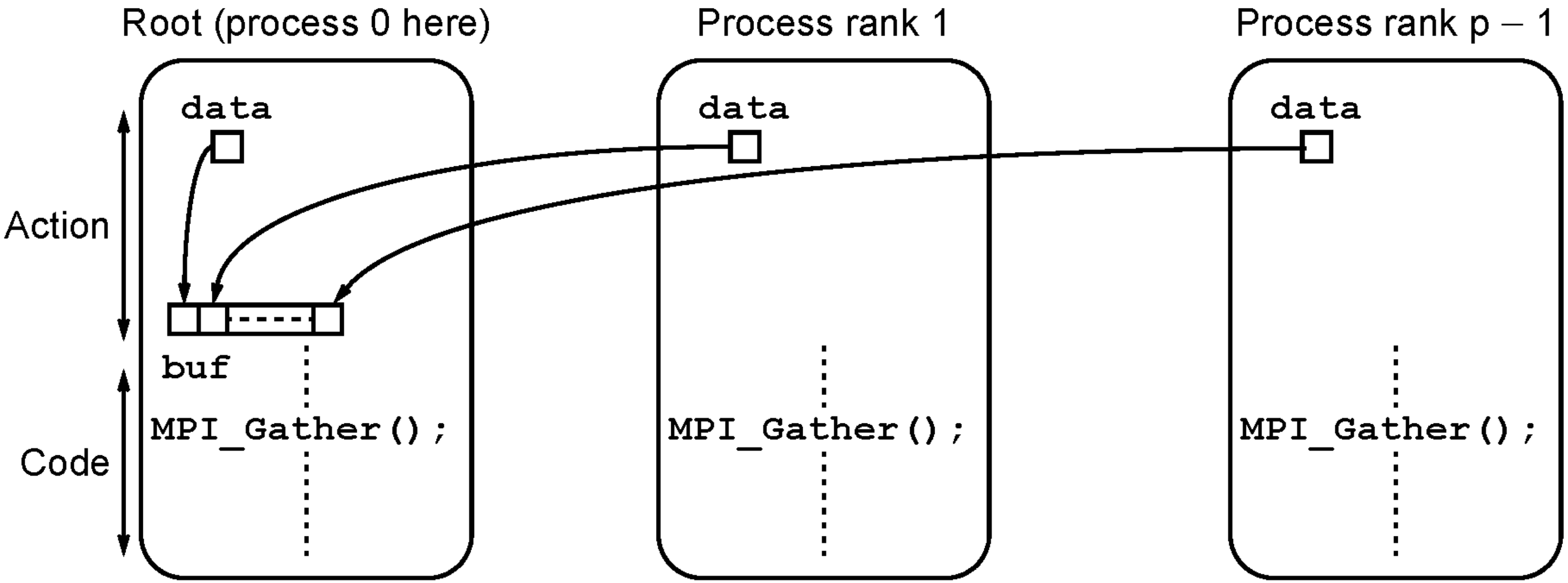
Example

In the following code, size of send buffer is given by 100 * <number of processes> and 100 contiguous elements are send to each process:

```
main (int argc, char *argv[]) {
    int size, *sendbuf, recvbuf[100];              /* for each process */
    MPI_Init(&argc, &argv);                        /* initialize MPI */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    sendbuf = (int *)malloc(size*100*sizeof(int));
        .
    MPI_Scatter(sendbuf,100,MPI_INT,recvbuf,100,MPI_INT,0,
                            MPI_COMM_WORLD);
        .
    MPI_Finalize();                                /* terminate MPI */
}
```

Gather

Having one process collect individual values from set of processes.



Gather parameters

```
MPI_Gather(*sbuf,scount,stype,*rbuf,rcount,rtype,root,comm)
```

Address of send buffer

Number of items to send *to root process*

Datatype of items sent

Address of receive buffer

Number of items to receive

Datatype of items received

Rank of root process (destination)

Communicator

Gather Example

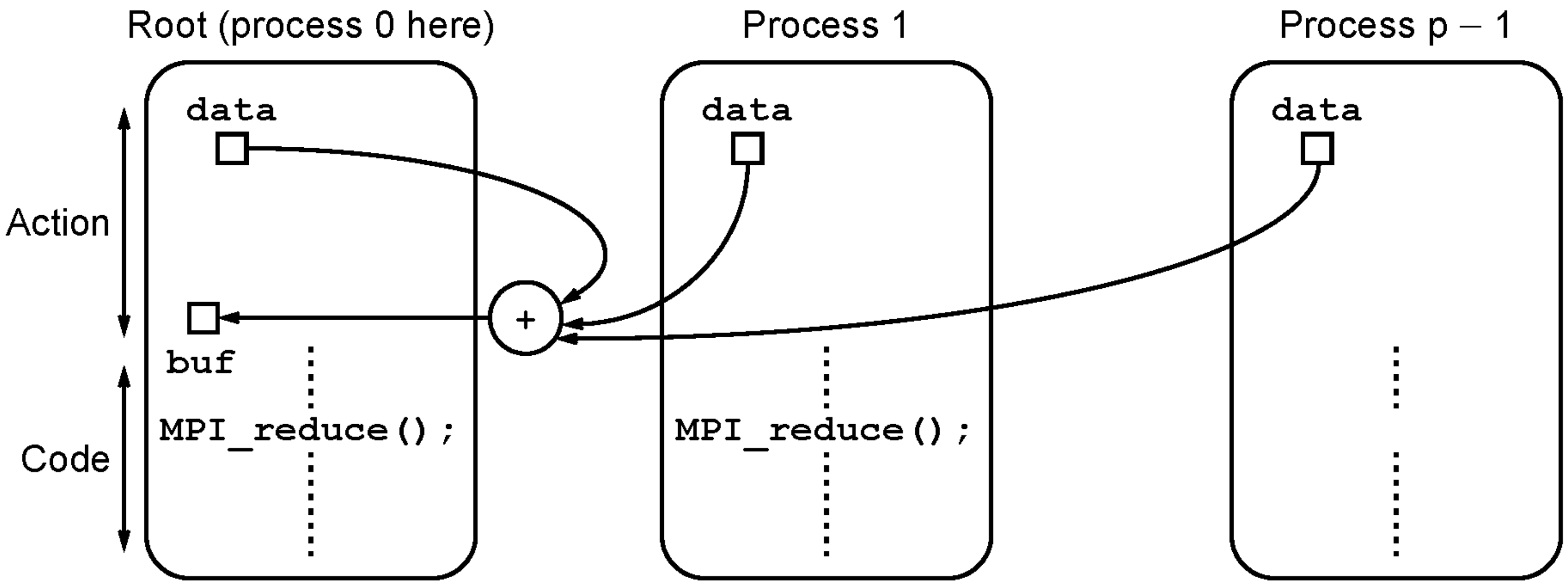
To gather items from group of processes into process 0, using dynamically allocated memory in root process:

```
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)); /*alloc. mem*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_COMM_
WORLD) ;
…
```

`MPI_Gather()` gathers from all processes, including root.

Reduce

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



Reduce parameters

```
MPI_Reduce(*sendbuf, *recvbuf, count, datatype, op root, comm)
```

Address of send buffer

Address of receive buffer

Number of items to send

Datatype of each item

Operation

Rank of root process (destination)

Communicator

Reduce - operations

`MPI_Reduce(*sendbuf,*recvbuf,count,datatype,op,root,comm)`

Parameters:

`*sendbuf`	send buffer address		
`*recvbuf`	receive buffer address		
`count`	number of send buffer elements		
`datatype`	data type of send elements		
`op`	reduce operation.		
	Several operations, including		
		`MPI_MAX`	Maximum
		`MPI_MIN`	Minimum
		`MPI_SUM`	Sum
		`MPI_PROD`	Product
`root`	root process rank for result		
`comm`	communicator		

Sample MPI program with collective routines

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000
void main(int argc, char *argv) {
    int myid, numprocs, data[MAXSIZE], i, x, low, high, myresult, result;
    char fn[255];
    char *fp;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    if (myid == 0) {  /* Open input file and initialize data */
            strcpy(fn,getenv("HOME"));
            strcat(fn,"/MPI/rand_data.txt");
            if ((fp = fopen(fn,"r")) == NULL) {
                    printf("Can't open the input file: %s\n\n", fn);
                    exit(1);
            }
            for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);
    }
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD); /* broadcast data */
    x = n/nproc; /* Add my portion Of data */
    low = myid * x;
    high = low + x;
    for(i = low; i < high; i++)
            myresult += data[i];
    printf("I got %d from %d\n", myresult, myid); /* Compute global sum */
    MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) printf("The sum is %d.\n", result);
    MPI_Finalize();
}
```

Collective routines
General features

- Performed on a group of processes, identified by a communicator
- Substitute for a sequence of point-to-point calls
- Communications are locally blocking
- Synchronization is *not* guaranteed (implementation dependent)
- Some routines use a *root* process to originate or receive all data
- Data amounts must exactly match
- Many variations to basic categories
- No message tags are needed

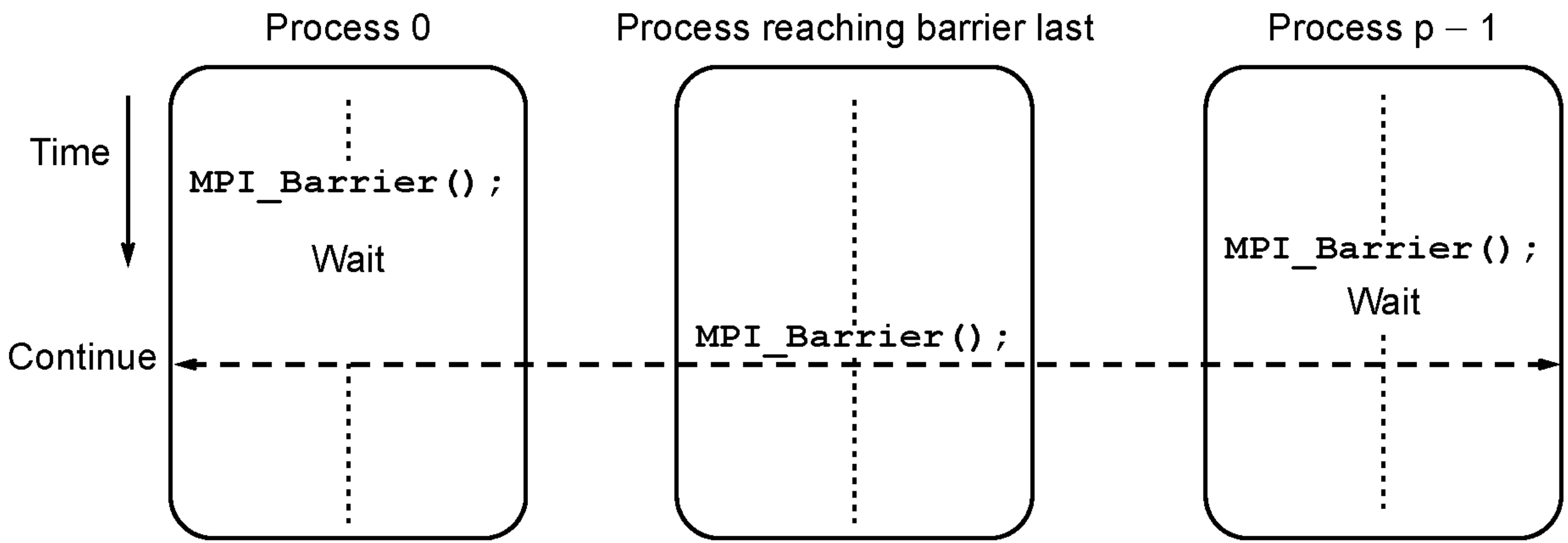
From http://www.pdc.kth.se/training/Talks/MPI/Collective.I/less.html#characteristics

Barrier

Block process until all processes have called it.
Synchronous operation.

`MPI_Barrier(comm)`

Communicator



Synchronous Message Passing

Routines that return when message transfer completed.

Synchronous send routine

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

  In MPI, MPI_SSend() routine.

Synchronous receive routine

- Waits until the message it is expecting arrives.

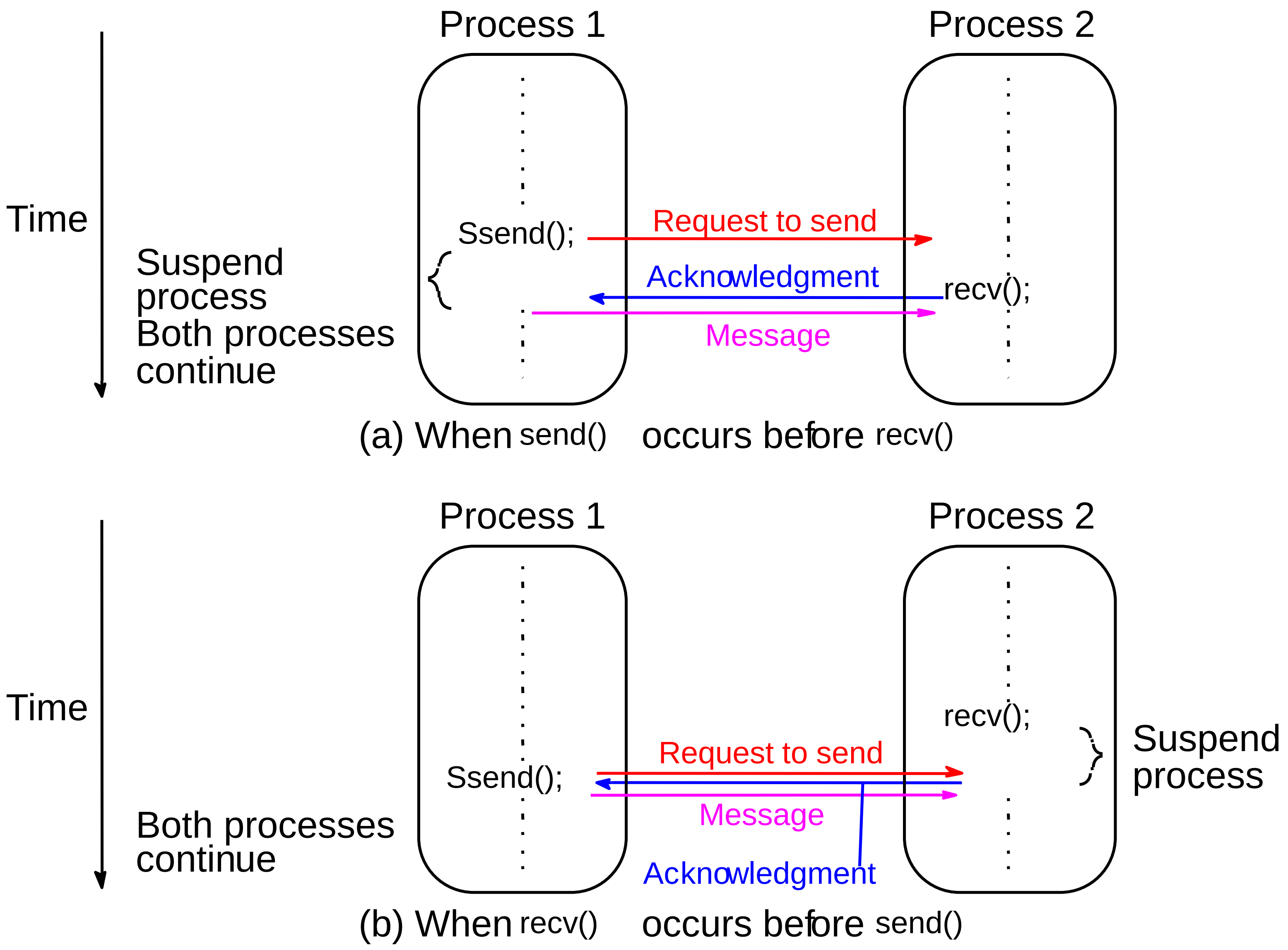
  In MPI, actually the regular MPI_recv() routine.

Synchronous Message Passing

Synchronous message-passing routines intrinsically perform two actions:

- They transfer data and
- They synchronize processes.

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



Parameters of synchronous send (same as blocking send)

MPI_Ssend(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

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. Sometimes called locally blocking.

- Non-blocking - return immediately (*asynchronous*)

Non-blocking 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.

Blocking/non-blocking terms may have different interpretations in other systems.

MPI Nonblocking Routines

- Non-blocking send - MPI_Isend() - will return “immediately” even before source location is safe to be altered.
- Non-blocking 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.

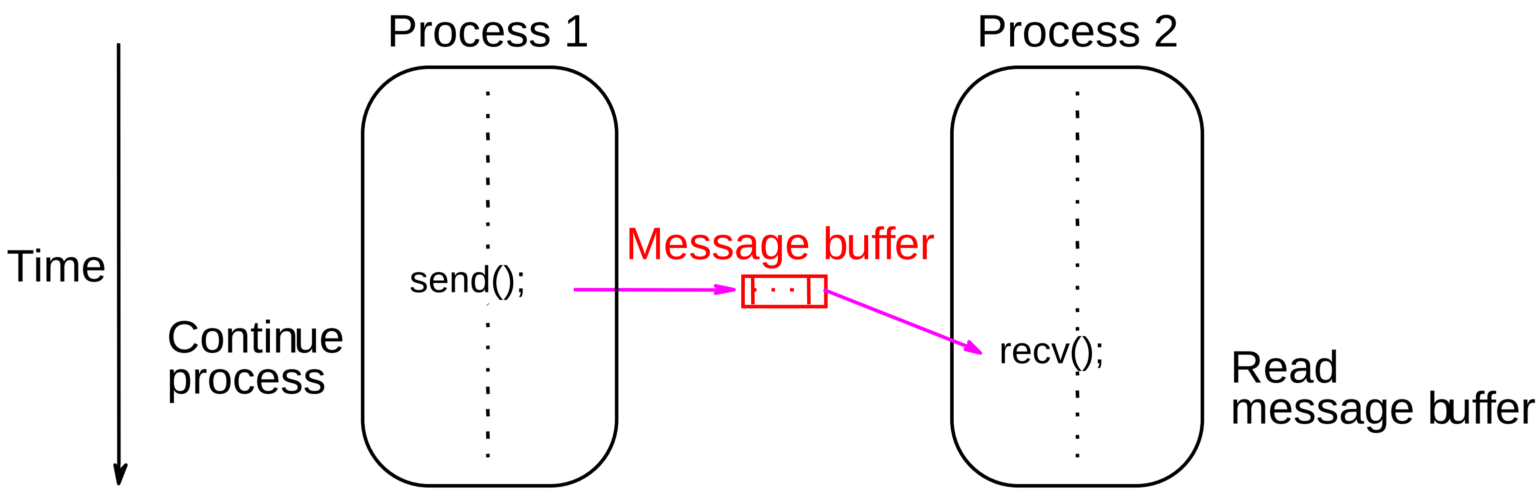
Example

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();
  MPI_Wait(req1, status);
} else if (myrank == 1) {
  int x;
  MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

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

- Message buffers only of finite length
- A point could be reached when send routine held up because all available buffer space exhausted.
- Then, send routine will wait until storage becomes re-available - i.e. routine will behave as a synchronous routine.