Lesson 2: Message-passing computing

G Stefanescu — University of Bucharest

Parallel & Concurrent Programming
Fall, 2014

Parallel programming options

Programming a message-passing multicomputer can be achieved by

- Designing a *special* parallel programming language (e.g., OCCAM for transputers)
- *Extending* the syntax/reserved words of an existing sequential high-level language to handle message passing (e.g., CC+, FORTRAN M)
- Using an existing sequential high-level language and providing a *library* of external procedures for message passing (e.g., MPI, PVM)

Another option will be to write a sequential program and to use a *parallelizing compiler* to produce a parallel program to be executed by multicomputer.

..Parallel programming options

We will concentrate on the third option. In such a case we have to say explicitly:

- *what processes* are to be executed
- *when to pass messages* between concurrent processes
- *what to pass* in the messages

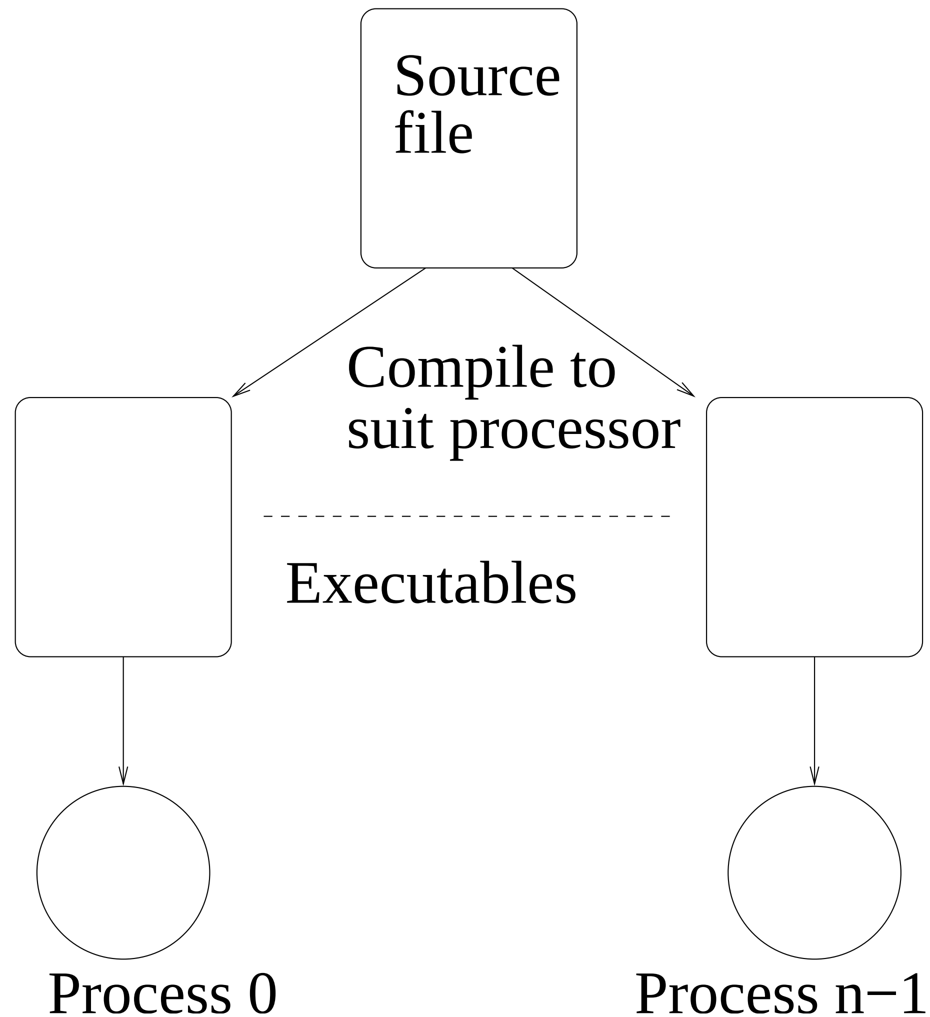
Two methods are needed for this form of message-passing systems:

- a method of *creating separate processes* for execution on different computers
- a method for *sending and receiving messages*

SPMD (Single Program Multiple Data) model

In this case *different processes are merged into one program*. Within the program there are control statements that will customize the code, i.e., select different parts for each process.

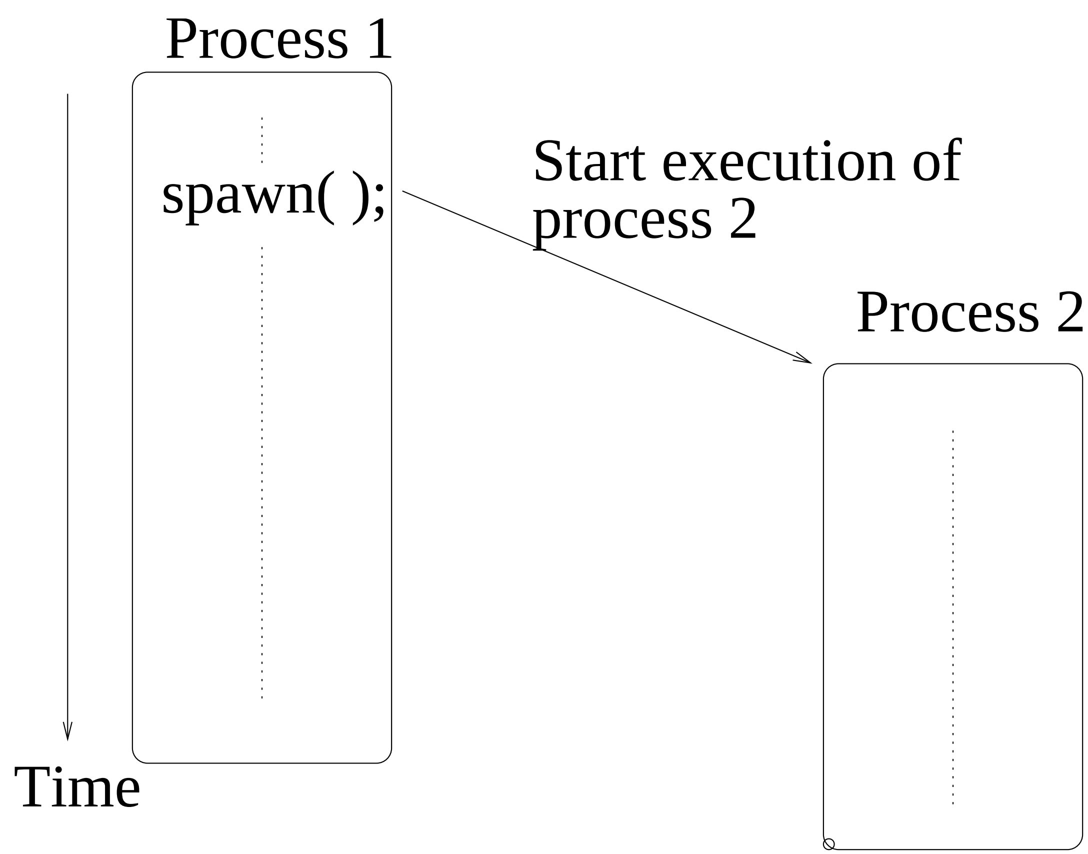
Basic features:
—usually *static* process creation
—a basic model is MPI



MPMD (Multiple Program Multiple Data) model

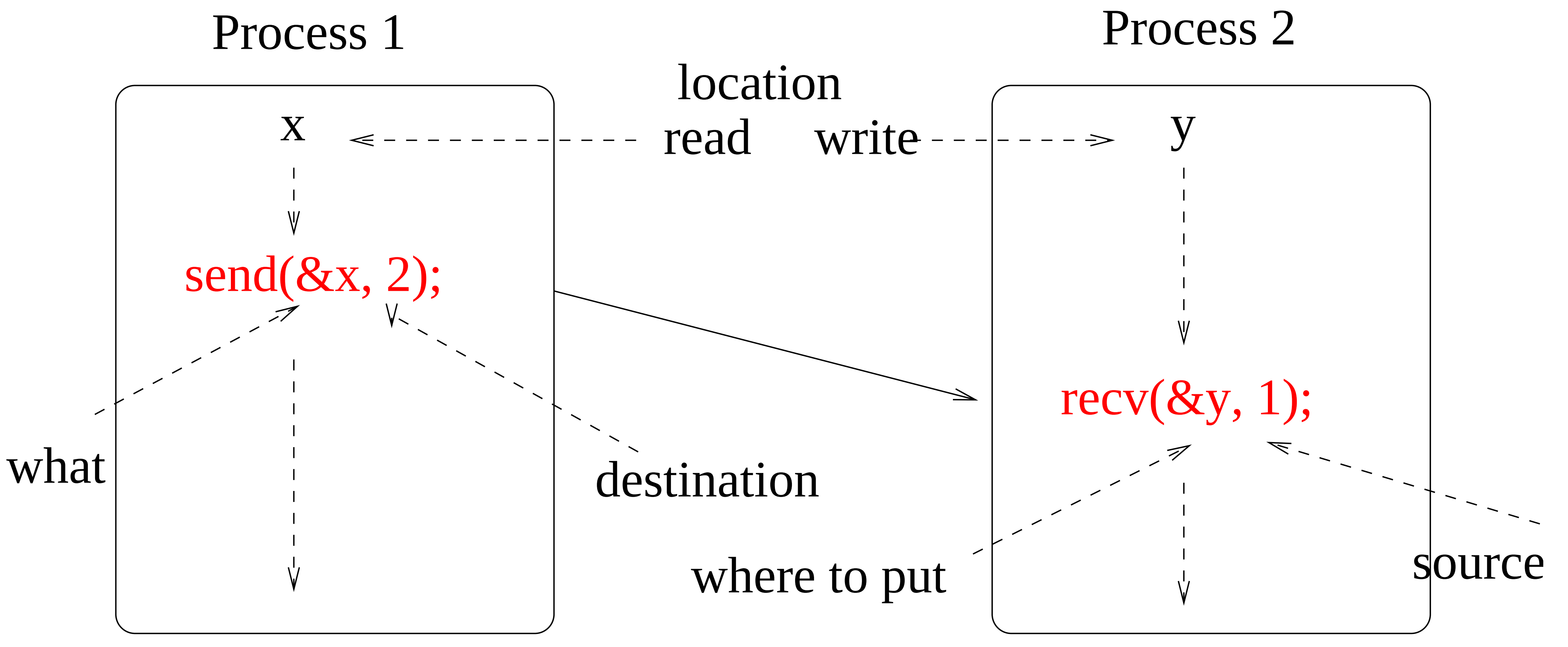
In this case *separate programs are written for each processor*. A *master-slave* approach is usually taken: a single processor executes a master process and the other processes are started from within the master process.

Basic features:
—usually *dynamic* process creation
—a basic model is PVM



Basic send and receive routines

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



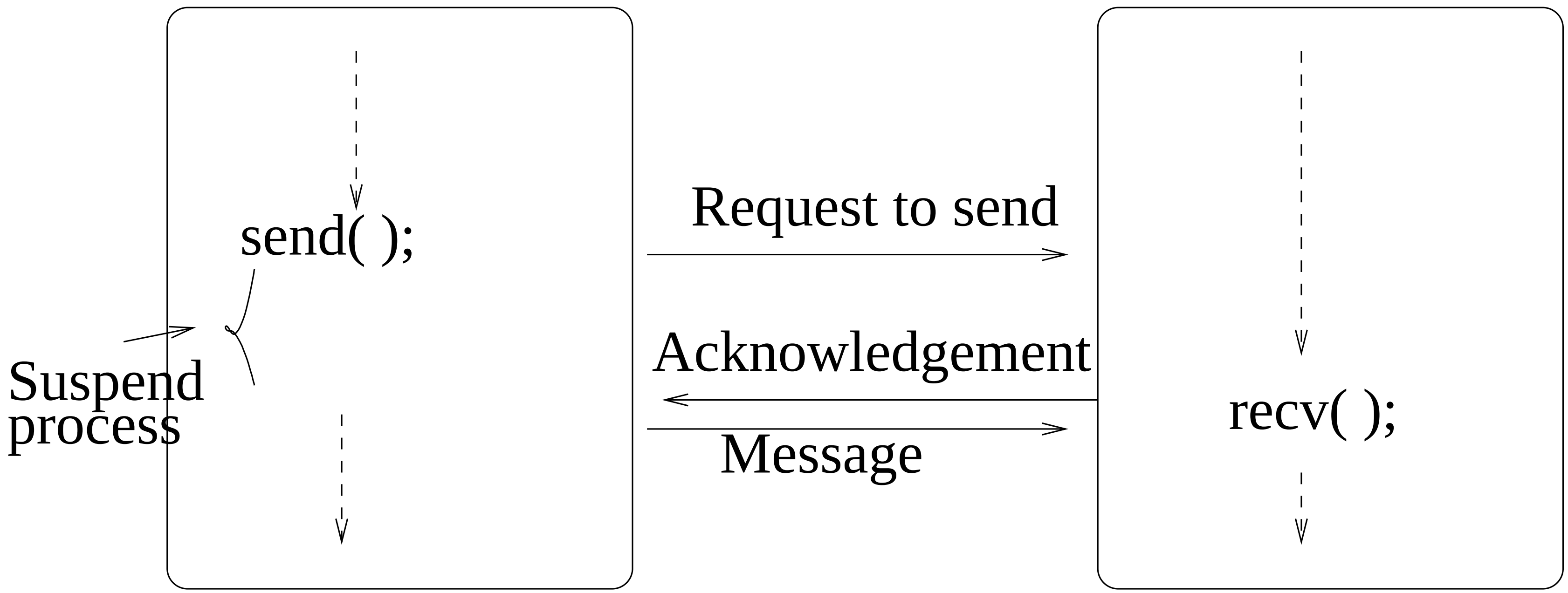
Synchronous message-passing

- *Synchronous message-passing routines return* when the message transfer has been completed.
- There is no need for message buffer storage.
  - The synchronous send routine could wait until the complete message can be accepted by the receiving process before sending the message.
  - The synchronous receive routine wait until the message it is expecting arrives.
- Synchronous routines perform two basic actions: They
  - *transfer data* and
  - *synchronize* processes

..Synchronous message-passing

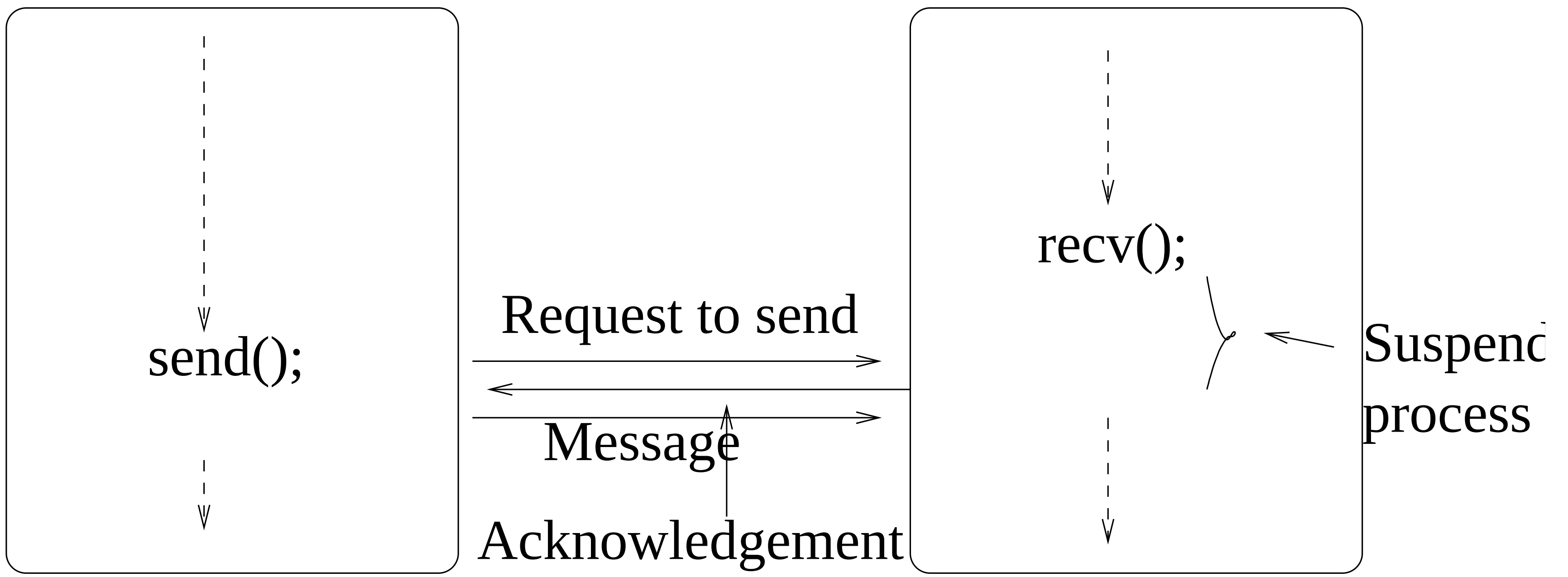
A three-way protocol is actually used here:

- Case 1: `Process 1` arrives to `send()` before `Process 2` arrives to `recv()`:



..Synchronous message-passing

- Case 2: `Process 1` arrives to `send()` after `Process 2` arrives to `recv()`:



Blocking and nonblocking message-passing

Blocking

- this term is used to describe routines that *do not return* until the transfer is *completed*.
- more precisely, the routines are *blocked from continuing* the process code
- generally speaking, the terms *synchronous* and *blocking* are synonymous

Non-blocking

- this term is used to describe routines that *return whether or not the message had been received*

Warring: These general terms were redefined in MPI, see below.

MPI definition of blocking and nonblocking

Blocking - return after their local actions are finished, though the message transfer may not have been completed (E.g., for `send()` it may return after the data are put in a buffer to be sent.)

Nonblocking - return immediately. In such a case it is assumes that the *data storage* to be used for the transfer *is not modified* by the subsequent statements before the transfer is completed and it is the programmer duty to ensure this.

Notice: This type of message passing is based on the use of message buffers *between the source and destination processes. As the buffers are of finite length, it may happen that the* `send()` *routine is blocked because the available buffer space has been exhausted.*

Message tag

A *message tag* is an extra information put in the message to differentiate between different messages being sent.

Example:

```
⋮                          ⋮
send(&x,2,5);              recv(&x,1,5);
⋮                          ⋮
(process 1)                (process 2)
```

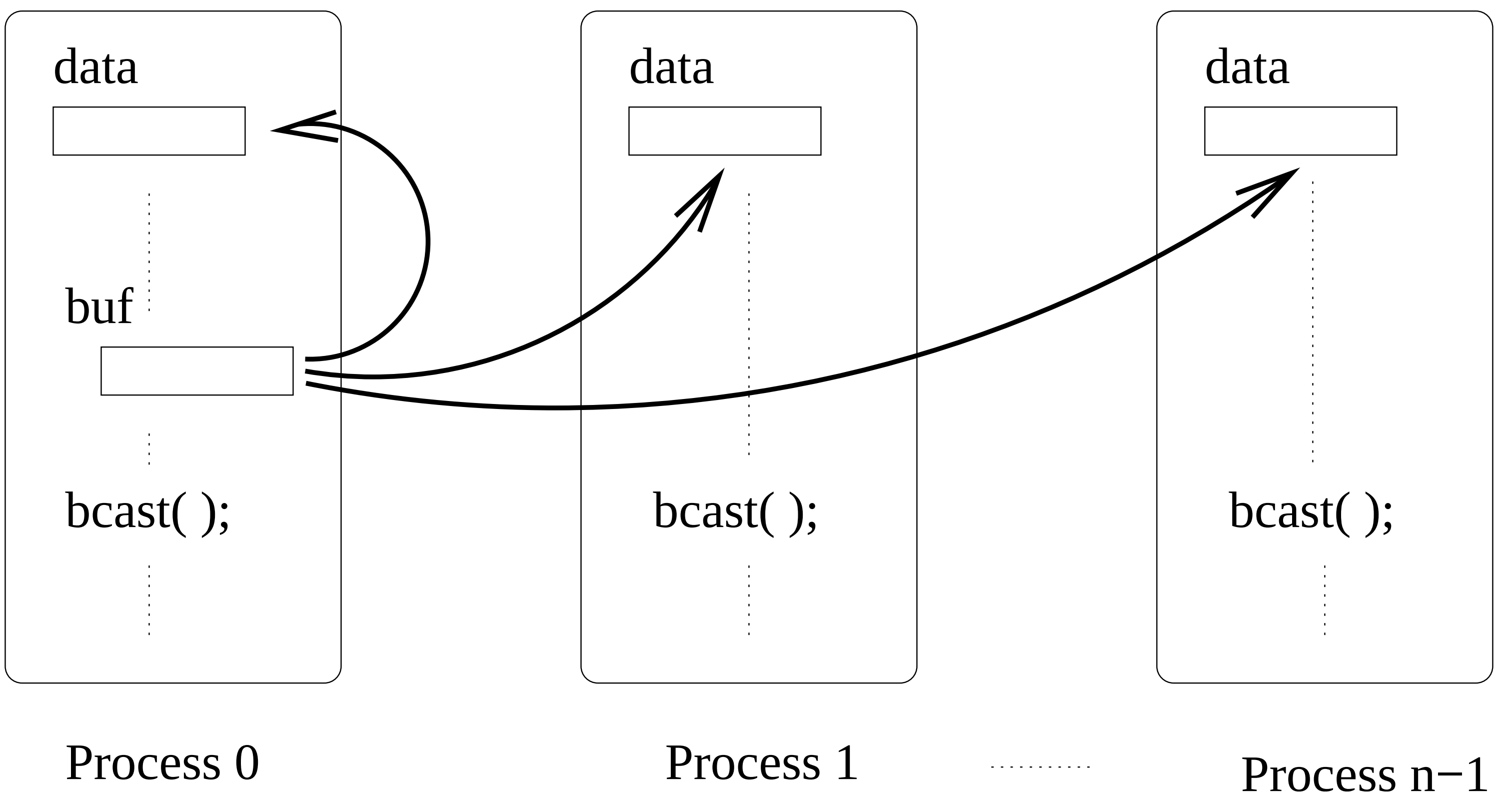
the tag 5 is used to match the send statement in `process 1` to the receive statement in `process 2`.

Notice: If such a special type matching is not required, then a *wild card* message tag is used, so that `recv()` will match *any* `send()`

Broadcast

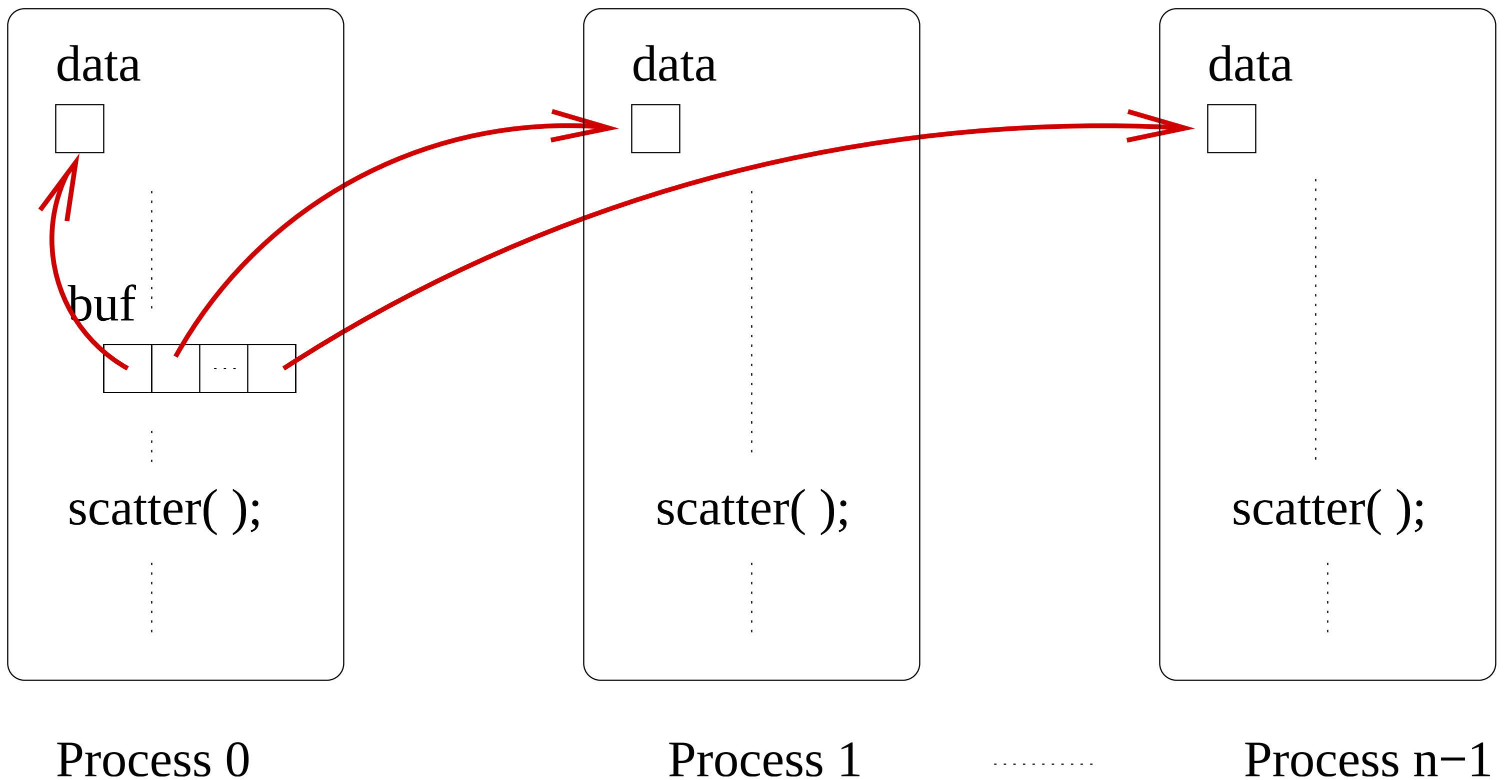
Broadcast is used to send the same message *to all* processes concerned with the problem.

Multicast is similar, but it is used to send a message *to a defined group* of processes.



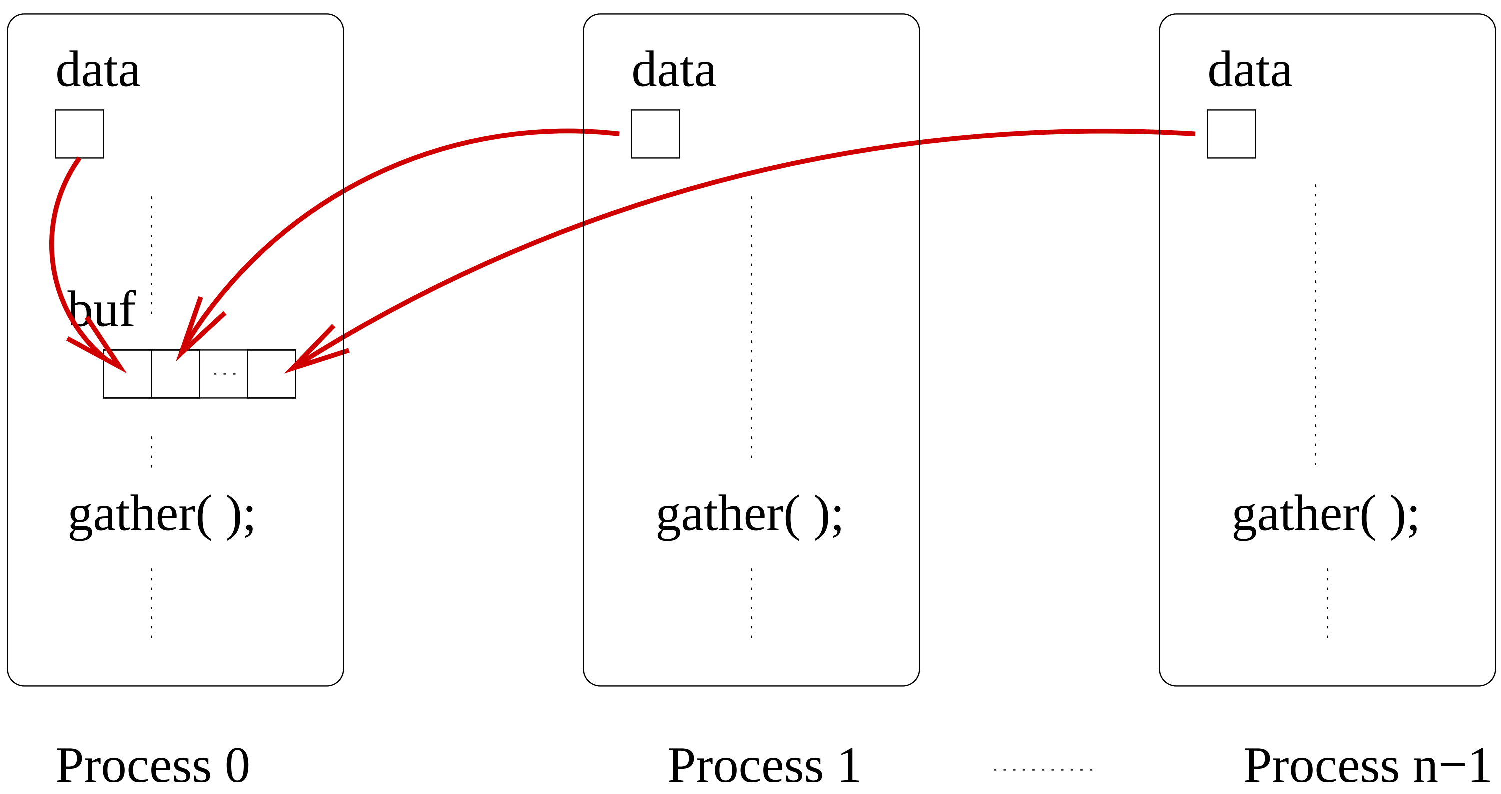
Scatter

Scatter is used to send *each element in an array* of data of the sending process to corresponding separate processes (datum from the i-th location goes to the i-th process).



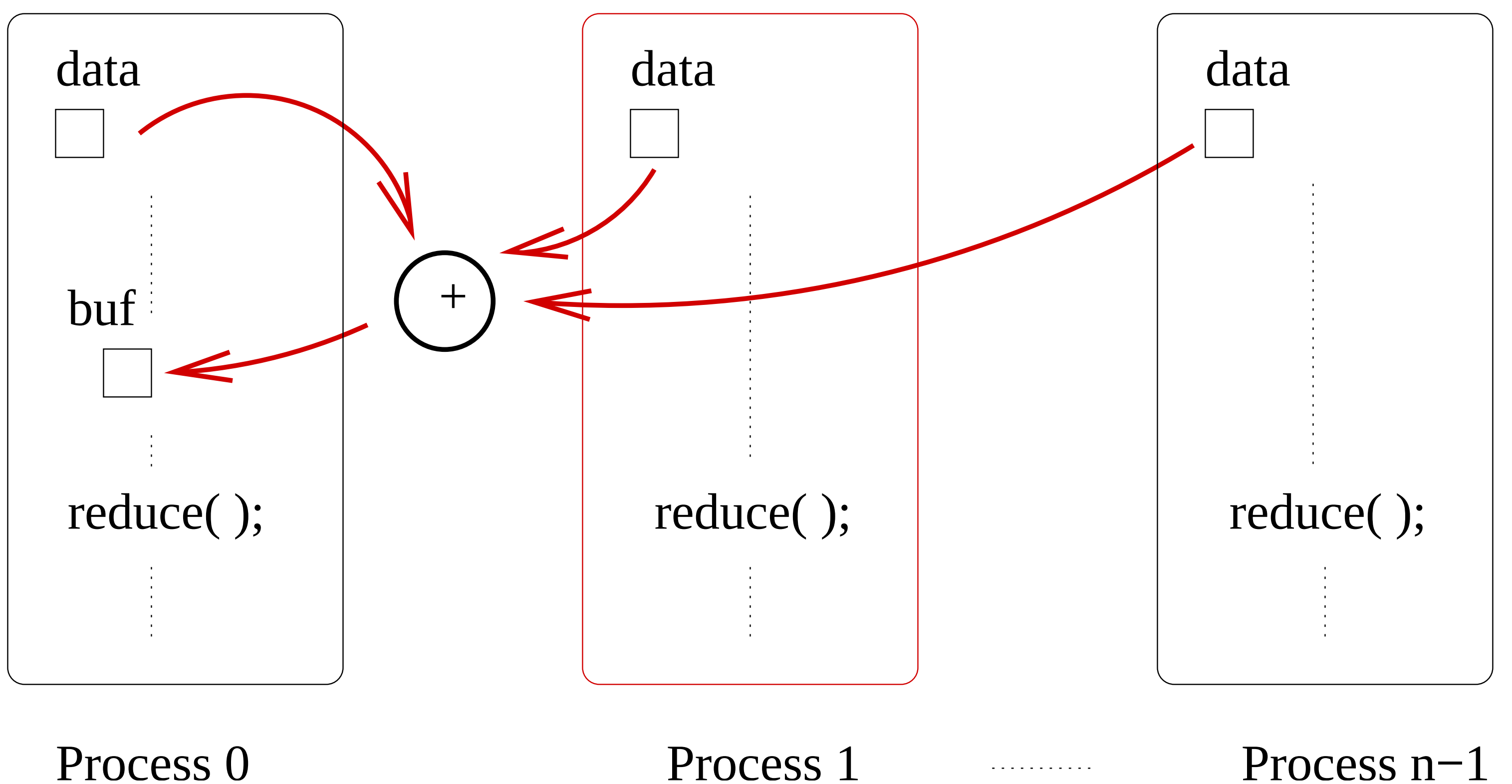
Gather

Gather is the opposite operation: the receiving process *collect in an array* the data sent by separate processes (datum from the i-th process goes to the i-th location).



Reduce

Reduce combines the `gather()` routine with an arithmetic or logical operation: the receiving process *collects* the data, *applies* the operation and *saves* it in its own memory.



Software tools

PVM (Parallel Virtual Machine)

- a first wildly accepted attempt to use a workstation cluster as a multicomputer.
- it may be used to run programs on both homogeneous or heterogeneous multicomputers
- it has a collection of library routines to be used with C or FORTRAN programs
- free

..Software tools

MPI (Message Passing Interface)

- it is a standard developed by a group of academics and industrial people to increase the use and the portability of message passing

- several free implementation exists [we use the one from Chicago, `mpich`]

PVM

- The programmer decomposes the problem into *separate programs*; each program is written in C (or FORTRAN) and compiled to be run on a specific type of computers in the network

- The *set of computers* used for the problem must be defined prior the running of the programs. (A convenient way to do this is by using a *host-file* listing the names of the computers available. This host-file is then read by PVM.)

- The *routing* of messages between computers is done by *PVM demon processes* installed by PVM on the computers that form the virtual machine.

Basic PVM message-passing routines

General:

- all PVM *send* routines are *nonblocking* (or *asynchronous*), while PVM *receive* routines are either blocking (or *synchronous*) or *nonblocking*.
- both *messages tags* and *wild cards* may be used to match send and the corresponding receive occurrences.

Basic PVM message-passing routines

Basic send-receive routines

- `pvm_psend()` and `pvm_precv()` - it has two goals: to pack data and to send them
- `pvm_send()` and `pvm_recv()` - send and receive without packing; it has to be used with explicit packing statements:
  —clear the buffer (`pvm_initsend()`) and pack the data (with `pvm_pkint()`, `pvm_pkstr()`, ...) before sending and
  —unpack the data (using `pvm_upkint()`, `pvm_upkstr()`, ...) after receiving
- broadcast, scatter, gather, reduce may all be used; the PVM statements are:
  `pvm_bcast()`,`pvm_scatter()`,`pvm_gather()`,`pvm_reduce()`

Example PVM program

We illustrate PVM programming with a simple *Sum* program: *the question is to add the numbers from a file using multiple processes.*

We use a master-slave approach:

- A master process creates the slave processes, reads data from the file and sends them to slaves (by multicast).
- Each slave identifies its portion of data, adds them and sends the result to the master [together with their identification number].
- The master receives the partial sums, adds them and prints the final result.

..Example PVM program

Master code

```
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>
#define SLAVE "spsum"
#define PROC 10
#define NELEM 1000
main(){
      int mytid,tids[PROC];
      int n = NELEM, nproc = PROC;
      int no,i,who,msgtype;
      int data[NELEM],result[PROC],tot=0;
      char fn[255];
      FILE *fp;
```

..Example PVM program

```
/* Start slave tasks */
        mytid = pvm_mytid(); /*enroll in PVM*/
        no = pvm_spawn(SLAVE,(char**)0,0,"",nproc,tids);
        if (no < nproc){
             printf("trouble spawning slaves \n");
             for (i=0; i<no; i++) pvm_kill(tids[i]);
             pvm_exit(); exit(1);
        }
/* Open input file and initialize data */
        strcpy(fn,getenv("HOME"));
        strcat(fn,"/pvm3/src/rand_data.txt");
        if ((fp = fopen(fn,"r")) == NULL){
             printf("Can't open input file %s\n",fn);
             exit(1);
        }
        for (i=0; i<n; i++)fscanf(fp,"%d",&data[i]);
```

..Example PVM program

```
/* Broadcast data to slaves */
        pvm_initsend(PvmDataDefault);
        msgtype = 0;
        pvm_pkint(&nproc,1,1);
        pvm_pkint(tids,nproc,1);
        pvm_pkint(&n,1,1);
        pvm_pkint(data,n,1);
        pvm_mcast(tids,nproc,msgtag);   ⟶ out.0
/* Get results from Slaves */
        msgtype = 5;
        for (i=0; i<nproc; i++){
                pvm_recv(-1,msgtype);   ⟵ in.5
                pvm_upkint(&who,1,1);
                pvm_upkint(&result[who],1,1);
                printf("%d from %d\n",result[who],who);
        }
```

..Example PVM program

```
/* Compute global sum */
        for (i=0; i<nproc; i++) tot += result[i];
        printf ("The total is %d.\n\n",tot);
/* Program finished; exit PVM */
        pvm_exit();
        return(0);
}
```

..Example PVM program

Slave code

```
#include <stdio.h>
#include "pvm3.h"
#define PROC 10
#define NELEM 1000
main(){
      int mytid,tids[PROC],n,me,i,msgtype;
      int x,nproc,master,data[NELEM],sum;
      mytid = pvm_mytid();
/* Receive data from master */
      msgtype = 0;
      pvm_recv(-1,msgtype);  ← in.0
      pvm_upkint(&nproc,1,1);
      pvm_upkint(tids,nproc,1);
      pvm_upkint(&n,1,1);
      pvm_upkint(data,n,1);
/* Determine my tid */
      for (i=0; i<nproc; i++)
           if (mytid == tids[i])
                {me = i; break;}
```

..Example PVM program

```
/* Add my portion of data */
      x = n / nproc;
      low = me * x;
      high = low + x;
      for (i=low; i<high; i++)
           sum += data[i];
/* Send result to master */
      pvm_initsend(PvnDataDeafult);
      pvm_pkint(&me,1,1);
      pvm_pkint(&sum,1,1);
      msgtype = 5;
      master = pvm_parent();
      pvm_send(master,msgtype);   ⟶ out.5
/* Exit PVM */
      pvm_exit();
      return(0);
}
```

MPI

General: MPI is a *standard* with various implementations; one writes a *single program*, each process running its own copy;

Process creation and execution: generally it is not defined; it is specified at compiling time how many processes are using; only static process creation is supported (in MPI, version 1)

Communications: one defines the *scope* of the communication operation; the set of all involved processes may be accessed using the predefined variable `MPI_COMM_WORLD`; each process has a unique rank, a number from 0 to $n-1$ (where n is the number of processes); other communication groups may be defined

..MPI

SPMD model: The shape of an MPI program is

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

..MPI

Global and local variables: By default, any global declaration of variables will be *duplicated* in each process; the variables that are not to be duplicated need to be declared within the code executed by that process

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0){
      int x,y;
⋮
} elseif (myrank == 1){
      int x,y;
⋮
}
```

(`x,y` from process 0 are different from `x,y` in process 1)

..MPI

Point-to-point communication: message tags and wild cards may be used (`MPI_ANY_TAG`, or `MPI_ANY_SOURCE`)

Blocking routines: return when they are locally complete, i.e., when the location used for the message can be used again without affecting the message being sent; general format:

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

where: `buf` - address of send buffer, `count` - number of items to send, `datatype` - datatype of each item, `dest` - rank of destination process, `tag` - message tag; `comm` - communicator

and

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

where: `buf` - address of receive buffer, `count` - maximum number of items to receive, `datatype` - datatype of each item, `src` - rank of source process, `tag` - message tag, `comm` - communicator, `status` - status after operation

..MPI

Example (blocking communication): To send an integer `x` from process 0 to process 1

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0){
       int x;
       MPI_Send(&x,1,MPI_INT,1,73,MPI_COMM_WORLD);
} elseif (myrank == 1){
       int x;
       MPI_Recv(&x,1,MPI_INT,0,73,MPI_COMM_WORLD,status);
}
```

..MPI

Non-blocking communication: `MPI_Isend()` and `MPI_Irecv()` - return "*i*mediatelly", even if the communication is not safe; to be used in combination with `MPI_Wait()` and `MPI_Test()` in order to ensure a complete communication.

Example (non-blocking communication): - same example

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0){
        int x;
        MPI_Isend(&x,1,MPI_INT,1,73,MPI_COMM_WORLD,req1);
        compute();
        MPI_Wait(req1, status);
} elseif (myrank == 1){
        int x;
        MPI_Recv(&x,1,MPI_INT,0,73,MPI_COMM_WORLD,status);
}
```

..MPI

Send communication modes: Four basic modes are

1. *Standard mode send* - it is not assumed that the corresponding receive routine has started (buffer space is not defined here; if buffering is provided, send can complete before the corresponding receive was reached)
2. *Buffered mode* - send may start and return before a matching receive was reached (here it is necessary to specify buffer space)
3. *Synchronous mode* - send and receive have to complete together (however, they may start at any time)
4. *Ready mode* - send can only start if a matching receive was already reached (use it with care...)

..MPI

Collective communication: This applies to processes included in a communicator. The main operations are:

`MPI_Bcast()` - broadcast from root to all other processes

`MPI_Gather()` - gather values from processes in the group

`MPI_Scather()` - scatter parts of the buffer to processes

`MPI_Alltoall()` - send data from all processes to all processes

`MPI_Reduce()` - collect and combine values from processes

`MPI_Reduce_scatter()` - combine values and scatter results

Barrier: May be used to synchronize processes by stopping each process until all have reached the barrier call

Example of MPI program

We illustrate MPI programming style with the same simple question: *add the numbers from a file using multiple processes.*

A similar master-slave approach is used:

- A master process (process 0) detects the number of processes from communicator, reads data from the file and sends them to all processes (by broadcast).
- Each process (including the master) identifies its portion of data and adds them.
- The master collects the partial sums and adds them (using `reduce` statement) and prints the final result.

..Example (MPI program)

```
#include "mpi.h"
#includes <stdio.h>
#include <math.h>
#define MAXSIZE 1000
void main(int argc, char *argv){
        int myid, numprocs;
        int 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){
                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);
                }
```

..Example (MPI program)

```
          for (i=0; i<MAXSIZE; i++) fscanf(fp,"%d",&data[i]);
     }
     /* Broadcast data */
     MPI_Bcast(data,MAXSIZE,MPI_INT,0,MPI_COMM_WORLD);
     /* Add my portion of data */
     x = n / nproc;
     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();
}
```

SoC Cluster

SoC has a Linux cluster *Tembusu*:

- a 64-node cluster of Dell PCs; each node has two 1.4GHz Intel PIII CPUs with 1GB of memory
- the nodes are connected by Myrinet as well as Gigabit ethernet
- besides the standard Linux suite of tools, MPI (versions using Myrinet and Gigabit ethernet) and PVM are available on the cluster.

See

`https://www.comp.nus.edu.sg/cf/tembusu/index.html`

for more.