Chapter 2

Message-Passing Computing

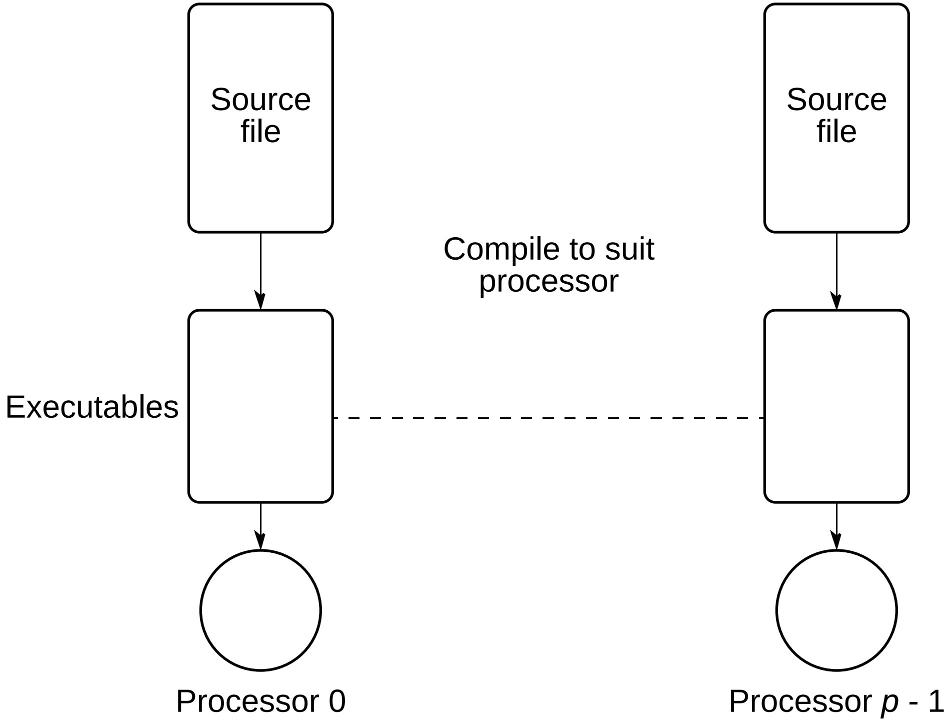
Basics of Message-Passing Programming using User-level Message Passing Libraries

Two primary mechanisms needed:

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

2. A method of sending and receiving messages

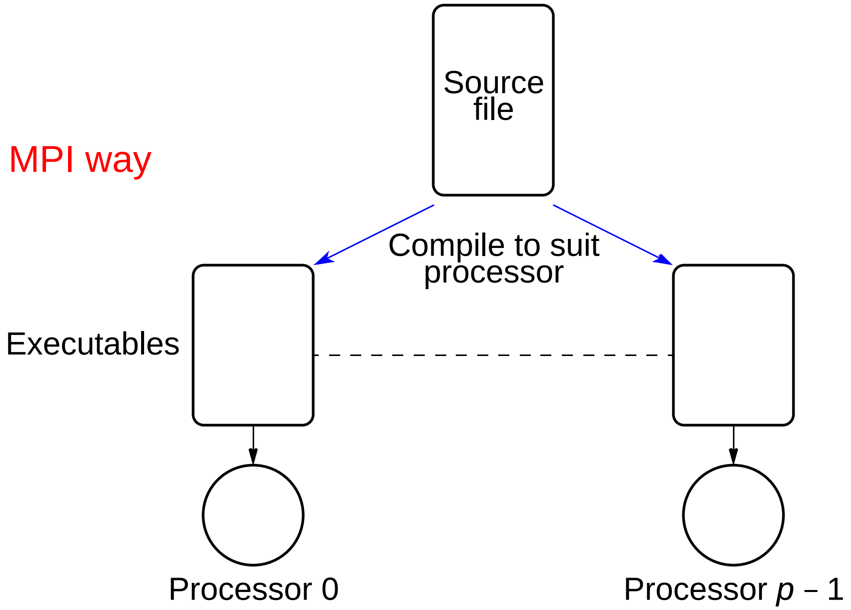
Multiple program, multiple data (MPMD) model



Single Program Multiple Data (SPMD) model

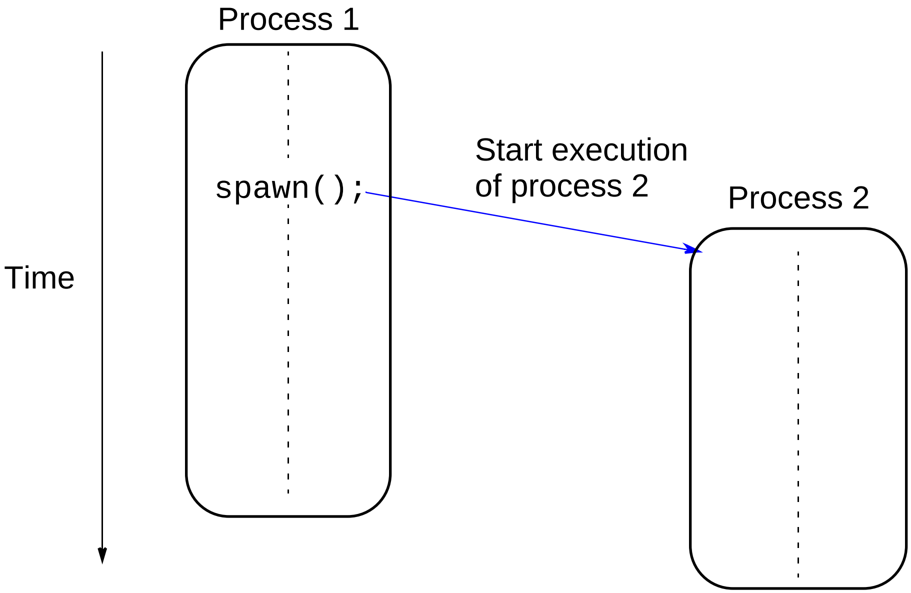
Different processes merged into one program. Within program, control statements select different parts for each processor to execute. All executables started together - static process creation.

Basic MPI way



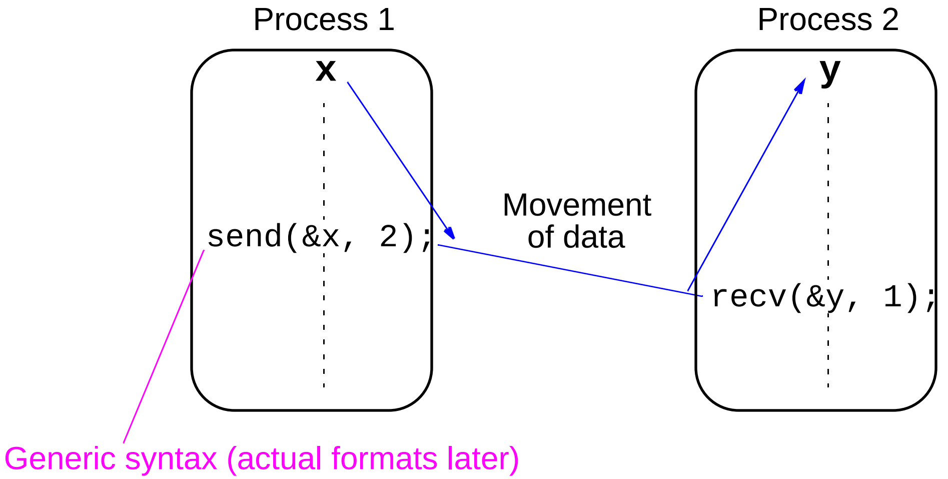
Multiple Program Multiple Data (MPMD) Model

Separate programs for each processor. Master-slave approach usually taken. One processor executes master process. Other processes started from within master process - dynamic process creation.



Basic “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

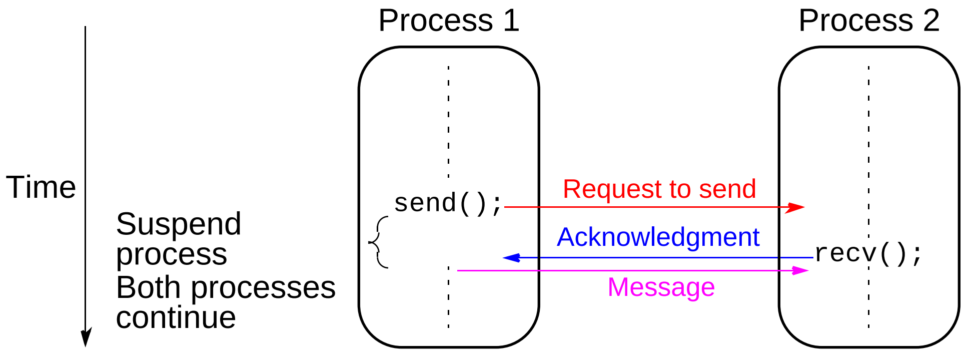
Waits until complete message can be accepted by the receiving process before sending the message.

Synchronous receive routine

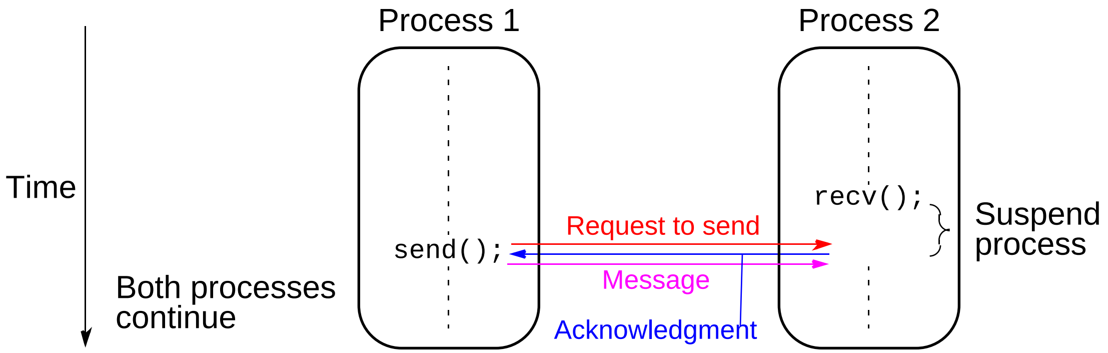
Waits until the message it is expecting arrives.

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

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



(a) When send() occurs before recv()



(b) When recv() occurs before send()

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.

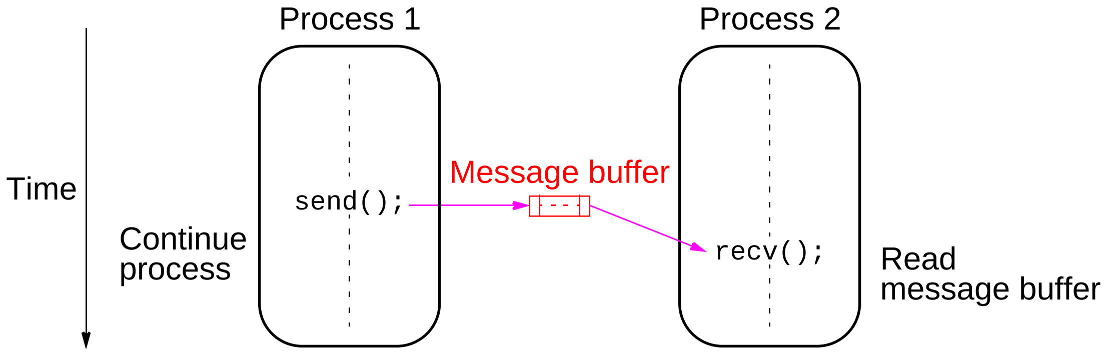
Non-blocking - return immediately.

Assumes that data storage to be used for transfer not modified by subsequent statements prior to tbeing 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 can 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 completed and message is safely on its way, sending process can continue with subsequent work.

Buffers only of finite length and 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 then routine behaves as a synchronous routine*.

Message Tag

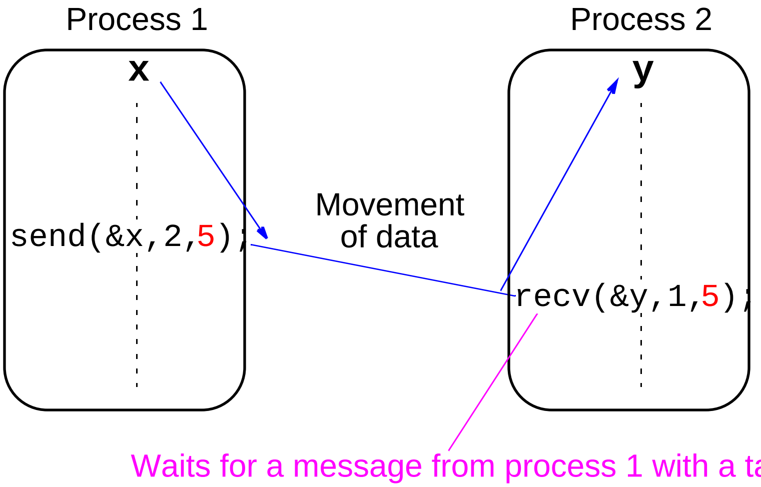
Used to differentiate between different types of messages being sent.

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()`**.

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:`**



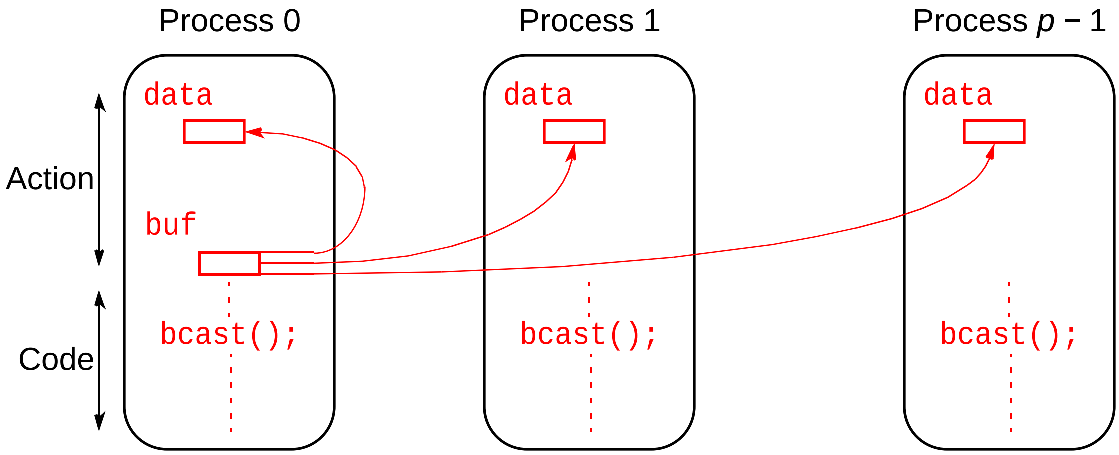
“Group” message passing routines

Apart from point-to-point 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.

Broadcast

Sending same message to all processes concerned with problem.

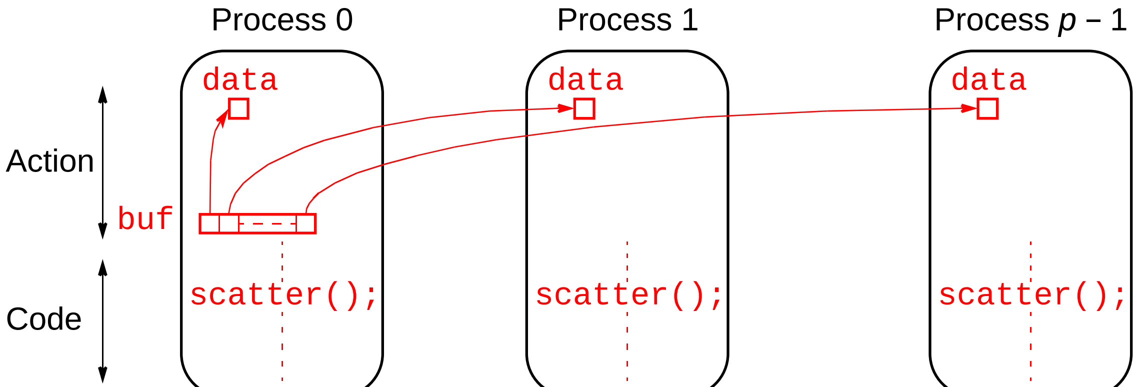
Multicast - sending same message to defined group of processes.



MPI form

Scatter

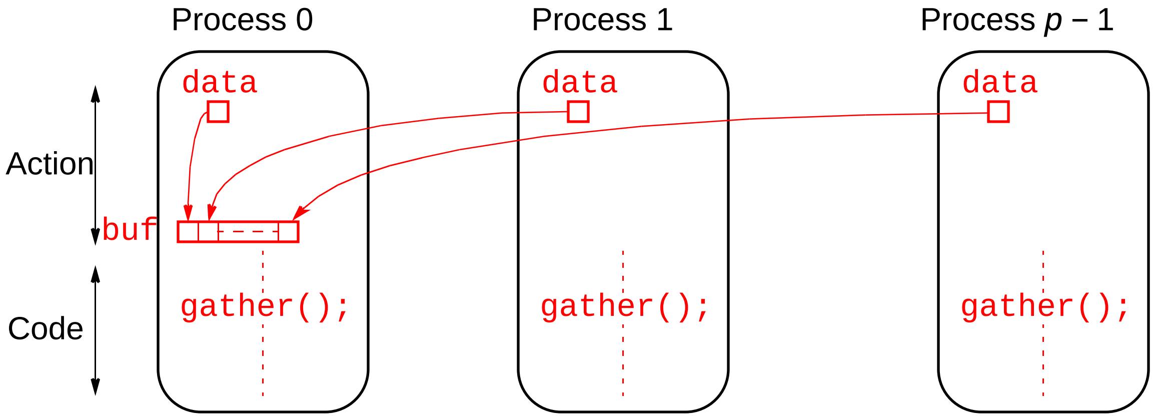
Sending each element of an array in *root* process to a separate process. Contents of *i*th location of array sent to *i*th process.



MPI form

Gather

Having one process collect individual values from set of processes.



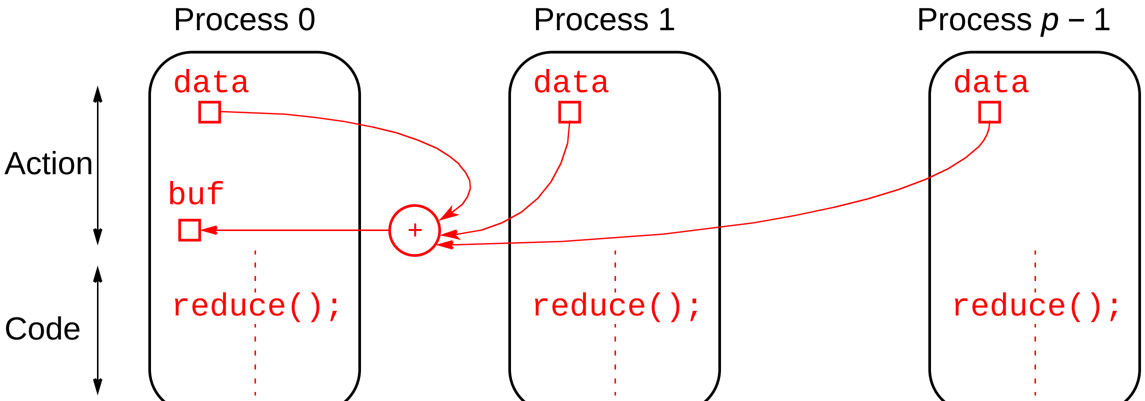
MPI form

Reduce

Gather operation combined with specified arithmetic/logical operation.

Example

Values could be gathered and then added together by root:



MPI form

PVM (Parallel Virtual Machine)

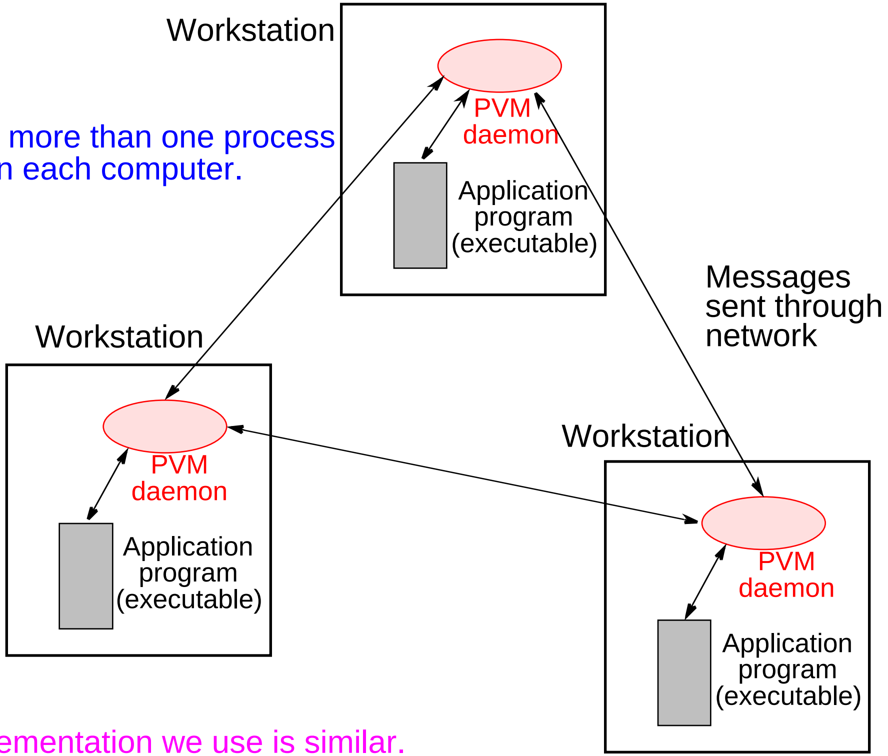
Perhaps first widely adopted attempt at using a workstation cluster as a multicomputer platform, developed by Oak Ridge National Laboratories. Available at no charge.

Programmer decomposes problem into separate programs (usually a master program and a group of identical slave programs).

Each program compiled to execute on specific types of computers.

Set of computers used on a problem first must be defined prior to executing the programs (in a hostfile).

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


Workstation
PVM daemon
Application program (executable)
Workstation
PVM daemon
Application program (executable)
Workstation
PVM daemon
Application program (executable)
Messages sent through network

Can have more than one process running on each computer.

MPI implementation we use is similar.

MPI (Message Passing Interface)

Standard developed by group of academics and industrial partners to foster more widespread use and portability.

Defines routines, not implementation.

Several free implementations exist.

MPI

Process Creation and Execution

Purposely not defined and will depend upon the implementation.

Only static process creation is supported in MPI version 1. All processes must be defined prior to execution and started together.

Orginally SPMD model of computation.

MPMD also possible with static creation - each program to be started together specified.

Communicators

Defines *scope* of a communication operation.

Processes have ranks associated with communicator.

Initially, all processes enrolled in a “universe” called **`MPI_COMM_WORLD`**, and each process is given a unique rank, a number from 0 to $p - 1$, where there are p processes.

Other communicators can be established for groups of processes.

Using the SPMD Computational Model

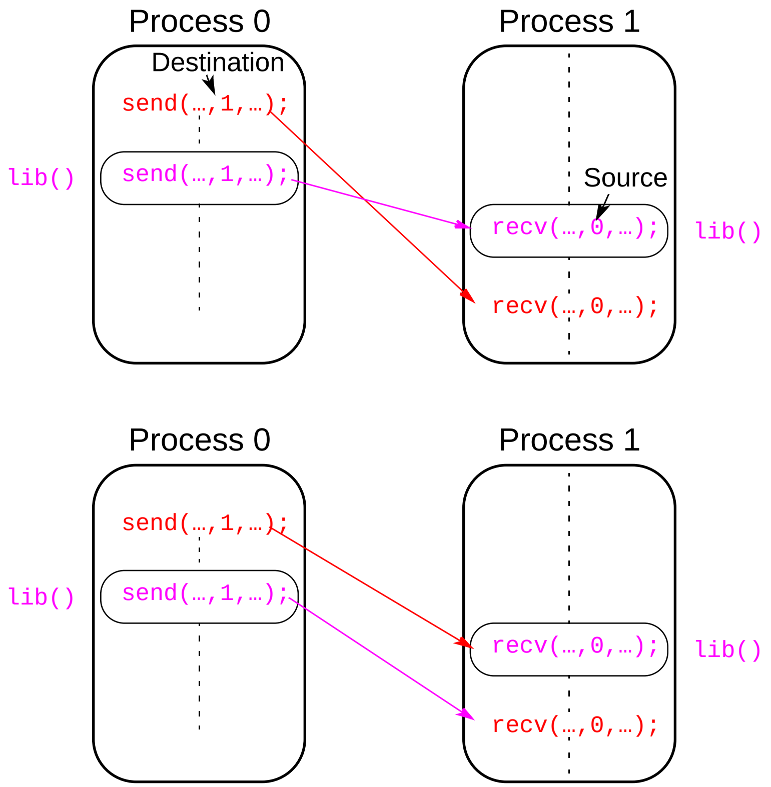
```
main (int argc, char *argv[])
{
MPI_Init(&argc, &argv);
.
.
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/*find process rank */
if (myrank == 0)
   master();
else
   slave();
.
.
MPI_Finalize();
}
```

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

“Unsafe” Message Passing

MPI specifically addresses unsafe message passing.

Unsafe message passing with libraries



MPI Solution

“Communicators”

A *communication domain* that defines a set of processes that are allowed to communicate between themselves.

The communication domain of the library can be separated from that of a user program.

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

Default Communicator

MPI_COMM_WORLD, exists as the first communicator for all the processes existing in the application.

A set of MPI routines exists for forming communicators.

Processes have a “rank” in a communicator.

Point-to-Point Communication

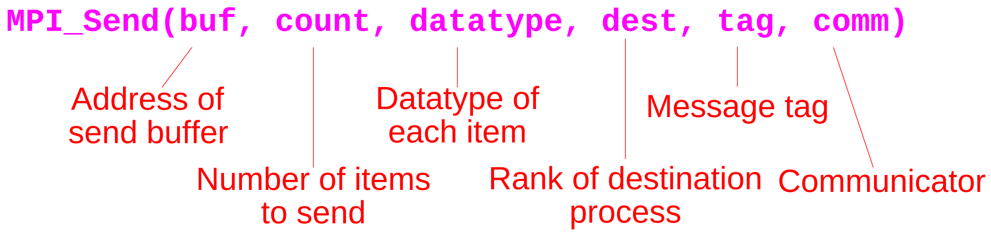
Uses send and receive routines with message tags (and communicator). Wild card message tags available

Blocking Routines

Return when they are locally complete - when location used to hold message can be used again or altered without affecting message being sent.

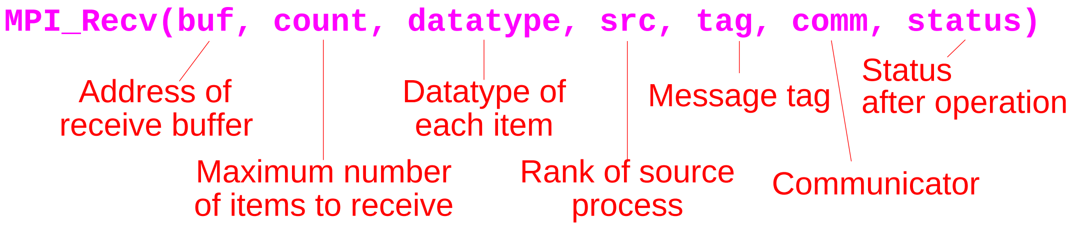
A blocking send will send the message and return. This does not mean that the message has been received, just that the process is free to move on without adversely affecting the message.

Parameters of the blocking send



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

Parameters of the blocking receive



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

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);
}
```

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 there is 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 the **`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);
}
```

Four 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.*

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

Involves set of processes, defined by an intra-communicator.

Message tags not present.

Broadcast and Scatter Routines

The principal collective operations operating upon data are

```
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 the group of processes into process 0, using dynamically allocated memory in the root process, we might use

```
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);
```

Note that **MPI_Gather()** gathers from all processes, including root.

Barrier

As in all message-passing systems, MPI provides a means of synchronizing processes by stopping each one until they all have reached a specific “barrier” call.

Sample MPI program.

```
#include “mpi.h”
#include <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) {           /* 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]);
   }
   /* 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();
}
```

Evaluating Parallel Programs

Equations for Parallel Execution Time

First concern is how fast parallel implementation is likely to be.

Might begin by estimating execution time on a single computer, t_s, by counting computational steps of best sequential algorithm.

For a parallel algorithm, in addition to number of computational steps, need to estimate communication overhead.

Parallel execution time, t_p, composed of two parts: a computation part, say t_{comp}, and a communication part, say t_{comm}; i.e.,

$$t_p = t_{\text{comp}} + t_{\text{comm}}$$

Computational Time

Can be estimated in a similar way to that of a sequential algorithm, by counting number of computational steps. When more than one process being executed simultaneously, count computational steps of most complex process. Generally, some function of *n* and *p*, i.e.

$$t_{\mathrm{comp}} = f(n, p)$$

The time units of t_p are that of a computational step.

Often break down computation time into parts. Then

$$t_{\mathrm{comp}} = t_{\mathrm{comp1}} + t_{\mathrm{comp2}} + t_{\mathrm{comp3}} + \ldots$$

where t_{comp1}, t_{comp2}, t_{comp3} … are computation times of each part.

Analysis usually done assuming that all processors are same and operating at same speed.

Communication Time

Will depend upon the number of messages, the size of each message, the underlying interconnection structure, and the mode of transfer. Many factors, including network structure and network contention. For a first approximation, we will use

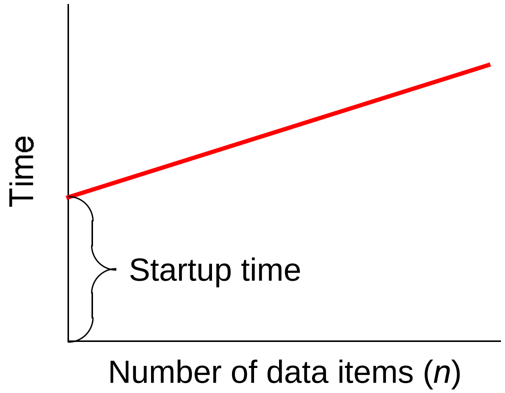
$$t_{\text{comm1}} = t_{\text{startup}} + nt_{\text{data}}$$

for communication time of message 1.

t_{startup} is the *startup time*, essentially the time to send a message with no data. Assumed to be constant.

t_{data} is the transmission time to send one data word, also assumed constant, and there are n data words.

Idealized Communication Time



Final communication time, t_{comm} will be the summation of communication times of all the sequential messages from a process, i.e.

$$t_{\text{comm}} = t_{\text{comm1}} + t_{\text{comm2}} + t_{\text{comm3}} + \ldots$$

Typically, the communication patterns of all the processes are the same and assumed to take place together so that only one process need be considered.

Both startup and data transmission times, t_{startup} and t_{data}, are measured in units of one computational step, so that we can add t_{comp} and t_{comm} together to obtain the parallel execution time, t_p.

Benchmark Factors

With t_s, t_{comp}, and t_{comm}, can establish speedup factor and computation/communication ratio for a particular algorithm/ implementation:

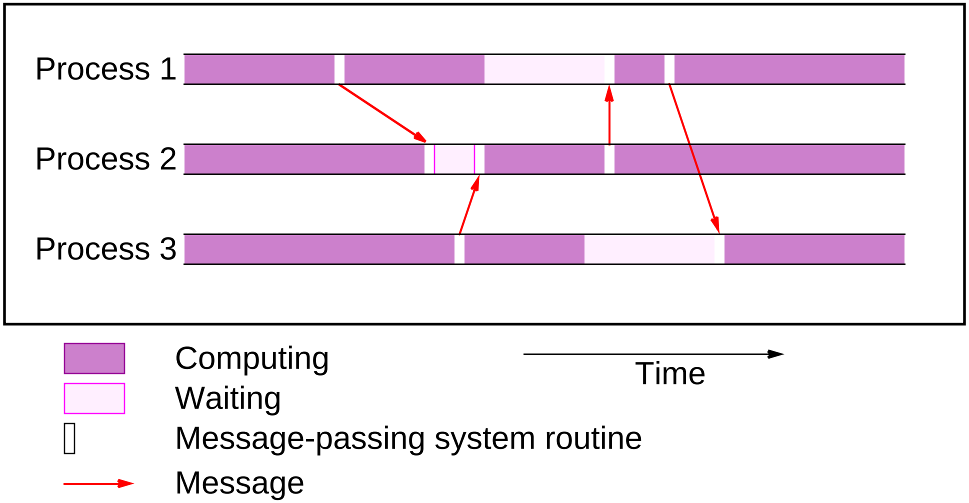
$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{t_s}{t_{comp} + t_{comm}}$$

$$\text{Computation/communication ratio} = \frac{t_{comp}}{t_{comm}}$$

Both functions of number of processors, p, and number of data elements, n.
Will give an indication of the scalability of the parallel solution with increasing number of processors and problem size. Computation/ communication ratio will highlight effect of communication with increasing problem size and system size.

Debugging and Evaluating Parallel Programs Empirically Visualization Tools

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



Implementations of visualization tools are available for MPI.

An example is the Upshot program visualization system.

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).

Parallel Programming Home Page

http://www.cs.uncc.edu/par_prog

Gives step-by-step instructions for compiling and executing programs, and other information.

Basic Instructions for Compiling/Executing MPI Programs

Preliminaries

- Set up paths
- Create required directory structure
- Create a file (hostfile) listing machines to be used (required)

Details described on home page.

Hostfile

Before starting MPI for the first time, need to create a hostfile

Sample hostfile

```
ws404
#is-sm1 //Currently not executing, commented
pvm1 //Active processors, UNCC sun cluster called pvm1 - pvm8
pvm2
pvm3
pvm4
pvm5
pvm6
pvm7
pvm8
```

Compiling/executing (SPMD) MPI program

For LAM MPI version 6.5.2. At a command line:

To start MPI:

First time: lamboot -v hostfile

Subsequently: lamboot

To compile MPI programs:

mpicc -o file file.c

or mpiCC -o file file.cpp

To execute MPI program:

mpirun -v -np no_processors file

To remove processes for reboot

lamclean -v

Terminate LAM

lamhalt

If fails

wipe -v lamhost

Compiling/Executing Multiple MPI Programs

Create a file specifying programs:

Example

1 master and 2 slaves, “appfile” contains

```
n0 master
n0-1 slave
```

To execute:

```
mpirun -v appfile
```

Sample output

```
3292 master running on n0 (o)
3296 slave running on n0 (o)
412 slave running on n1
```