Chapter 2

Message-Passing Computing

Basics of Message-Passing Programming using user-level message passing libraries

Two primary mechanisms needed:

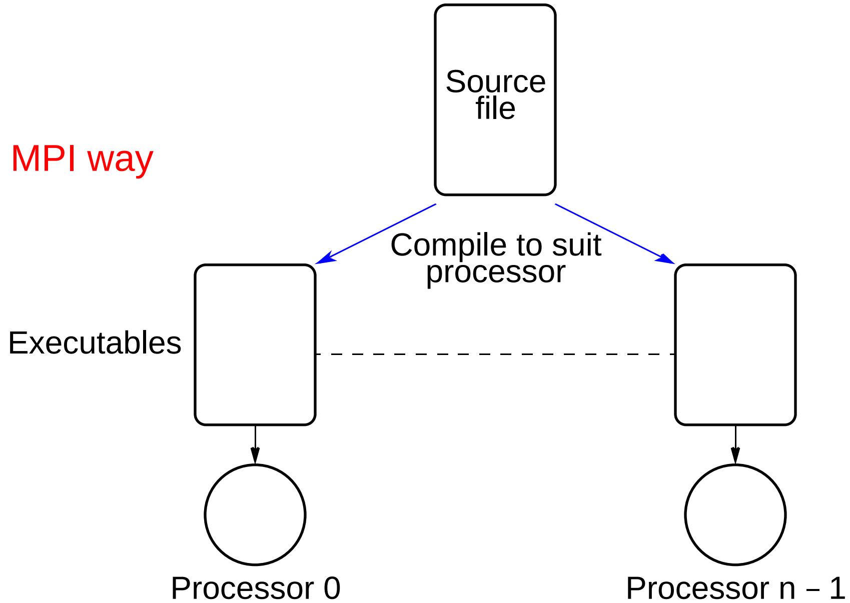
1. A method of creating separate processes for execution on different computers

2. A method of sending and receiving messages

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

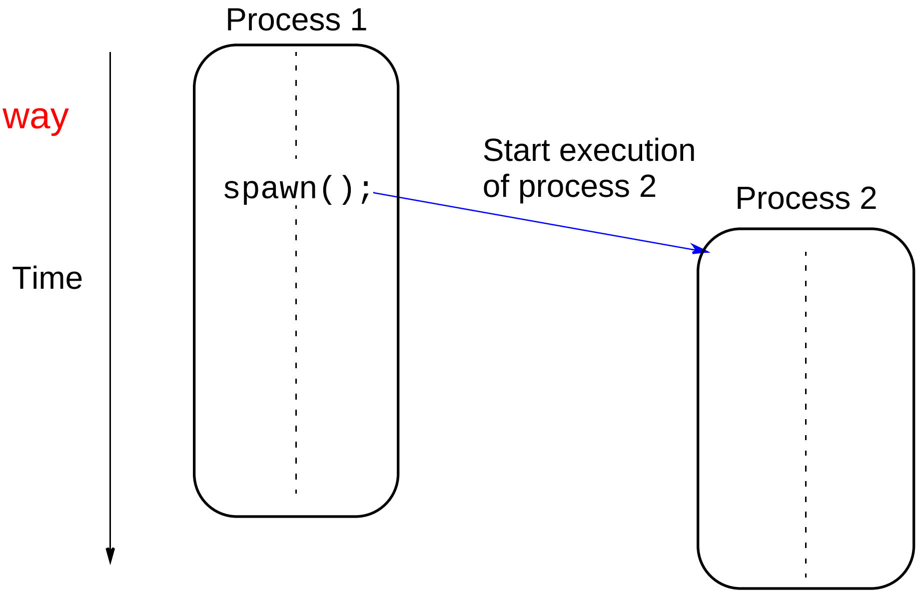


Slides for *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers* by Barry Wilkinson and Michael Allen, Prentice Hall Upper Saddle River New Jersey, USA, ISBN 0-13-671710-1.

Multiple Program Multiple Data (MPMD) Model

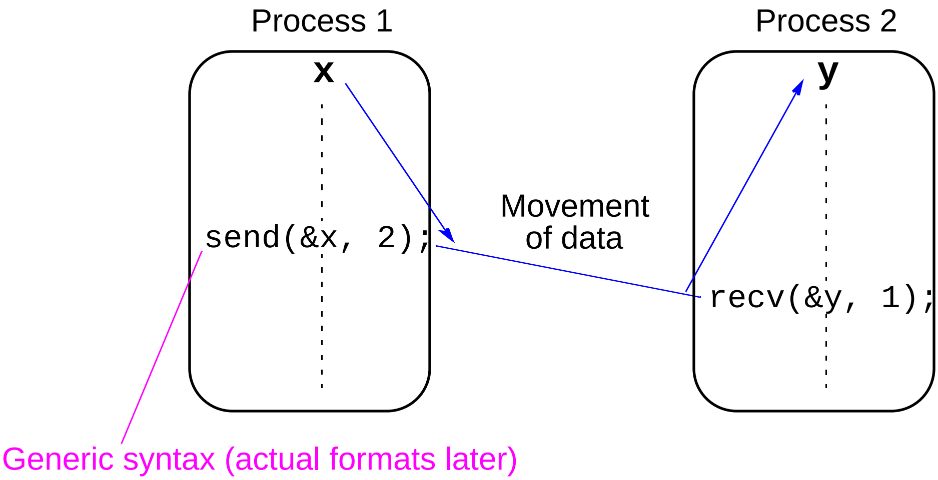
Separate programs for each processor. Master-slave approach usually taken. One processor executes master process. Other processes started from within master process - dynamic process creation.

PVM way



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

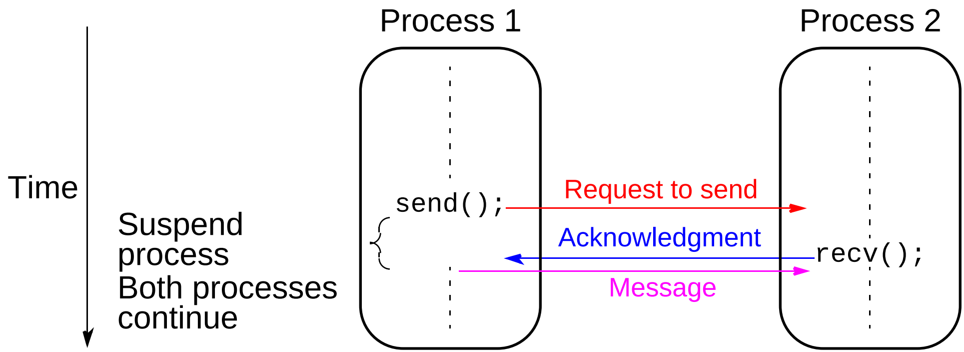
Waits until complete message can be accepted by the receiving process before sending the message.

Synchronous receive routine

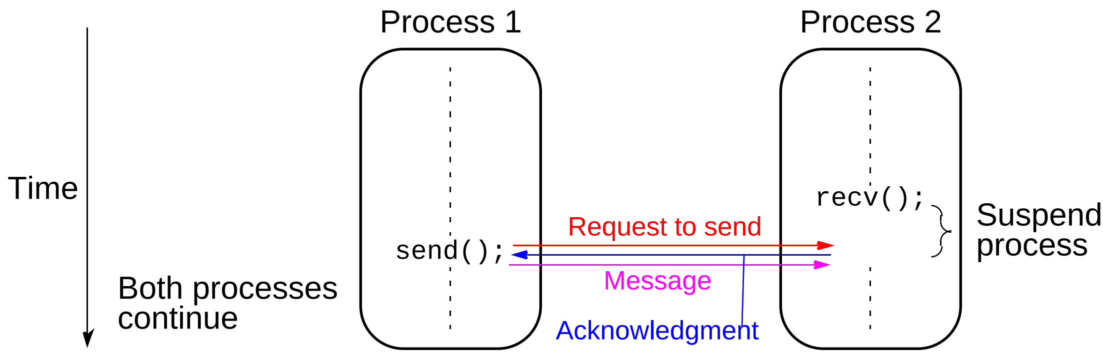
Waits until the message it is expecting arrives.

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.

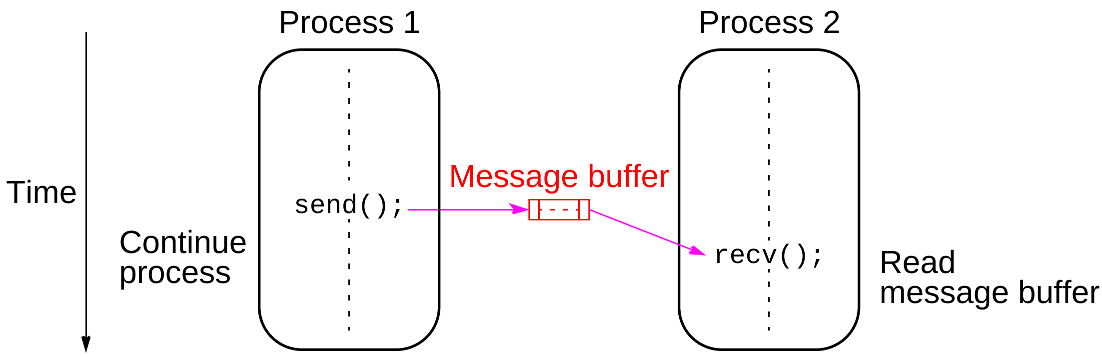
Non-blocking - return immediately.

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.

Notices 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

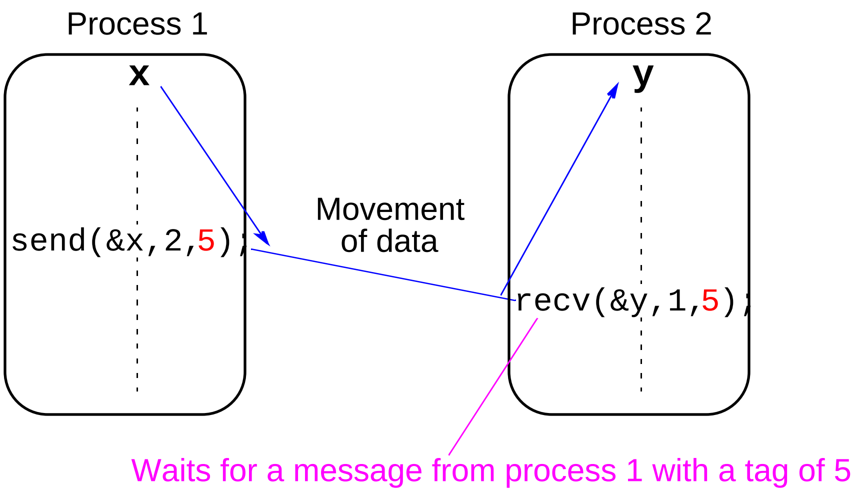
Used to differentiate between different types of messages being sent.

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:**



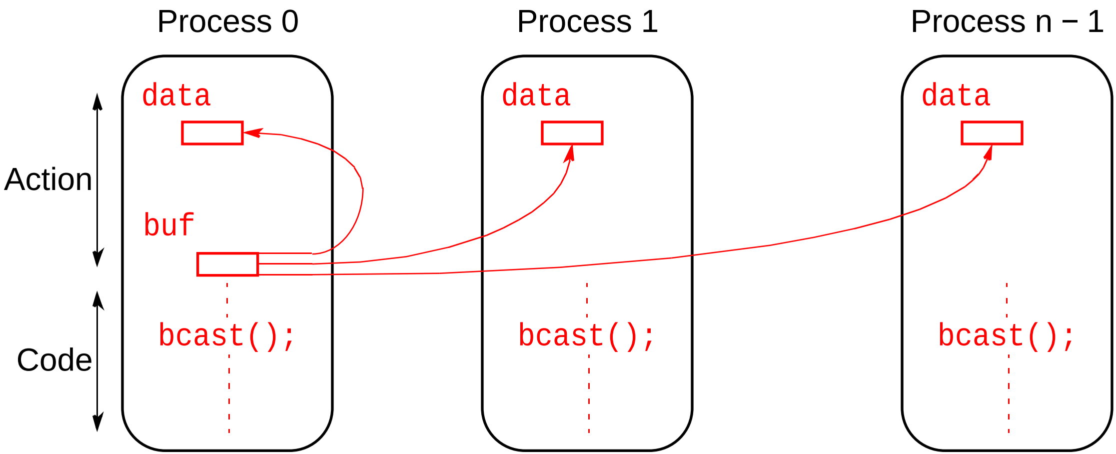
“Group” message passing routines

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.

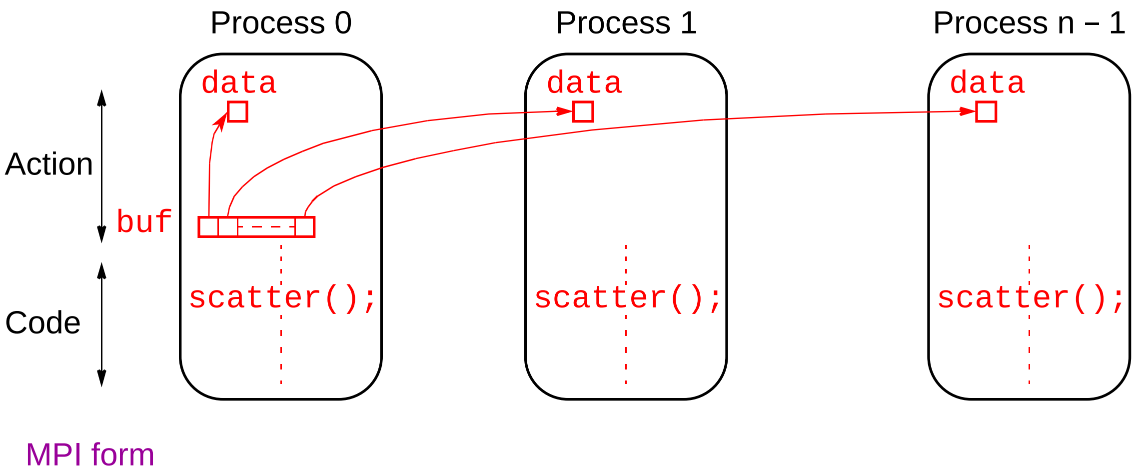
Multicast - sending same message to defined group of processes.



MPI form

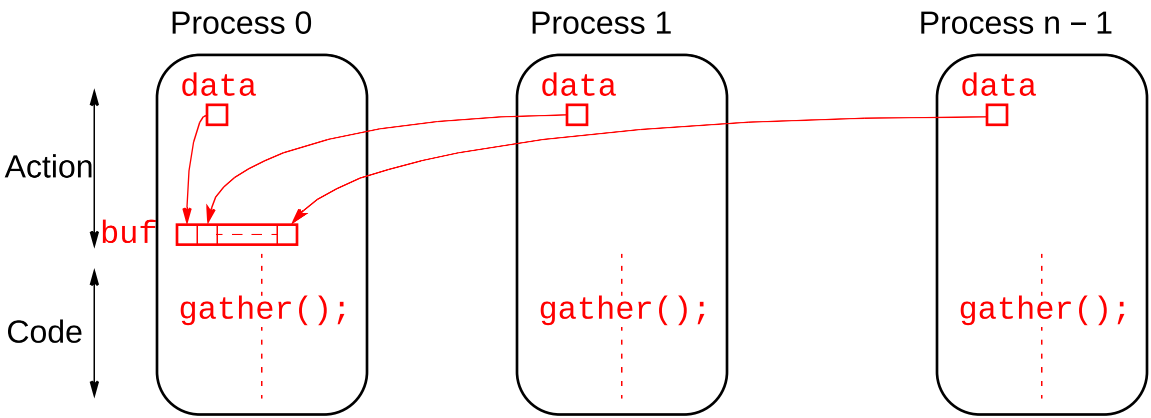
Scatter

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.



Gather

Having one process collect individual values from set of processes.



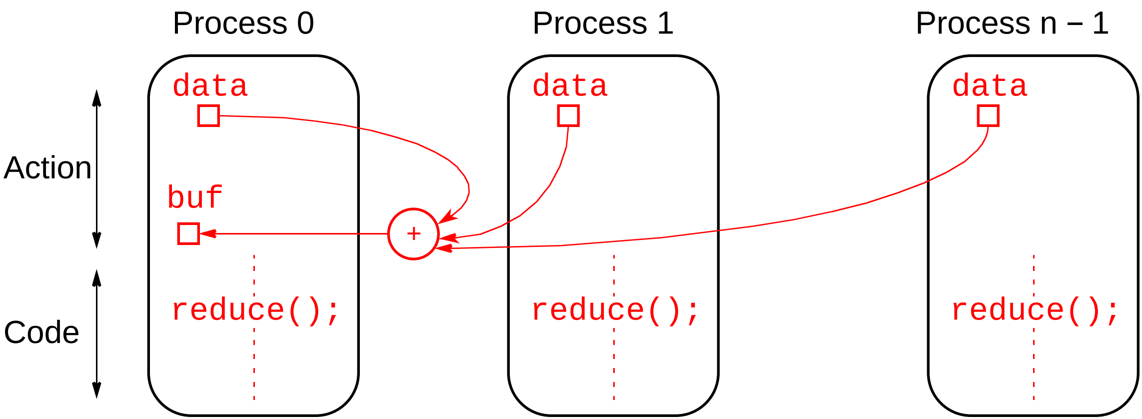
MPI form

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.

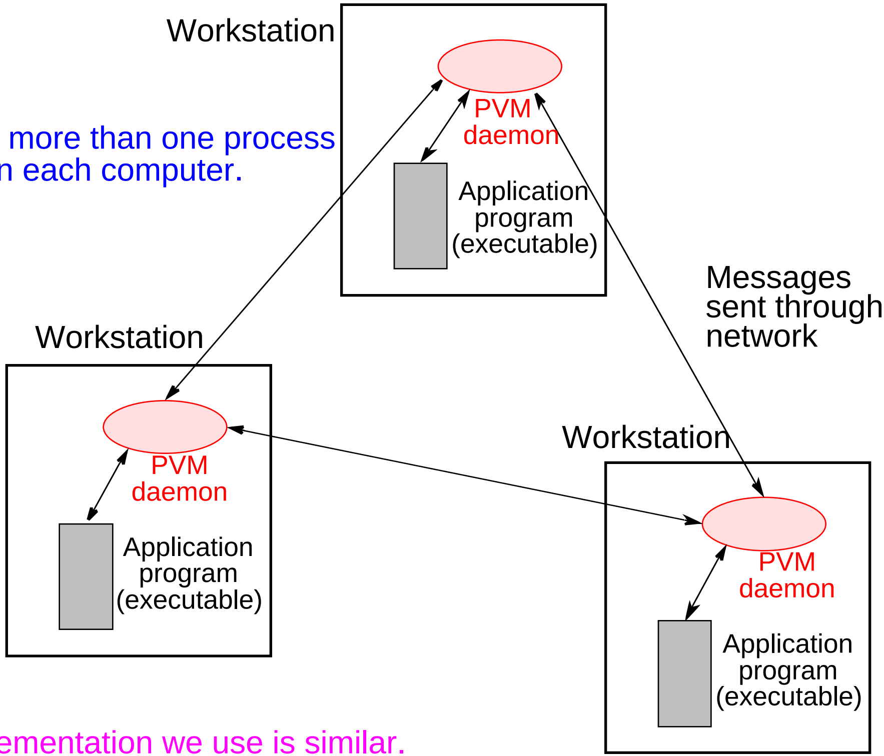
Programmer decomposes problem into separate programs (usually a master program and a group of identical slave programs).

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*.

Can have more than one process running on each computer.



MPI implementation we use is similar.

Slides for *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers* by Barry Wilkinson and Michael Allen, Prentice Hall Upper Saddle River New Jersey, USA, ISBN 0-13-671710-1.

PVM Message-Passing Routines

All PVM send routines are nonblocking (or asynchronous in PVM terminology)

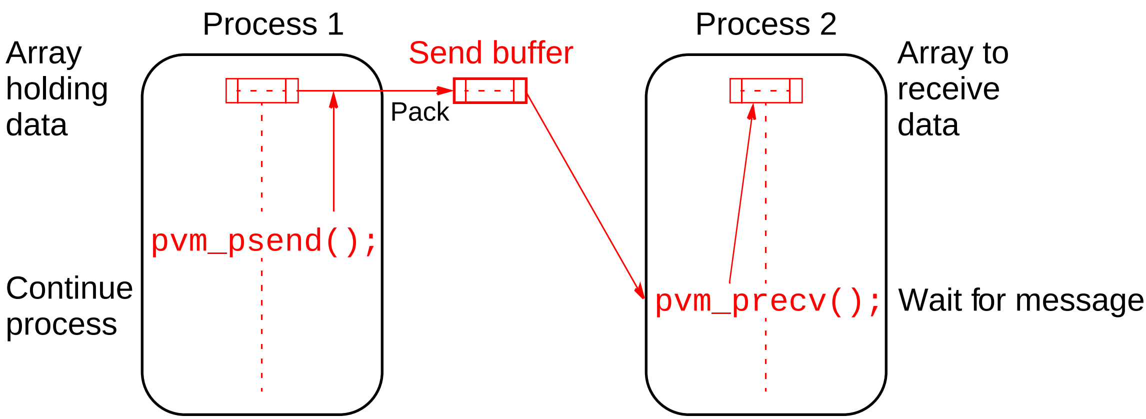
PVM receive routines can be either blocking (synchronous) or nonblocking.

Both message tag and source wild cards available.

Basic PVM Message-Passing Routines

pvm_psend()and pvm_precv()system calls.

Can be used if data being sent is a list of items of the same data type.



Slides for *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers* by Barry Wilkinson and Michael Allen, Prentice Hall Upper Saddle River New Jersey, USA, ISBN 0-13-671710-1.

Full list of parameters for `pvm_psend()` and `pvm_precv()`

```
pvm_psend(int dest_tid, int msgtag, char *buf, int len, int datatype)

pvm_precv(int source_tid, int msgtag, char *buf, int len, int datatype)
```

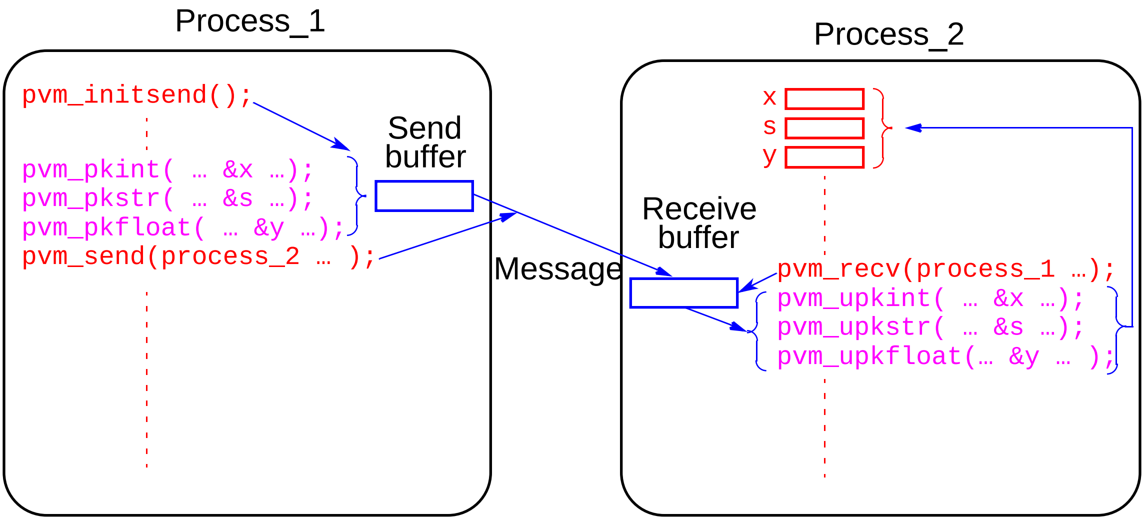
Sending Data Composed of Various Types

Data packed into send buffer prior to sending data.

Receiving process must unpack its receive buffer according to format in which it was packed.

Specific packing/unpacking routines for each datatype.

Sending Data Composed of Various Types Example



Broadcast, Multicast, Scatter, Gather, and Reduce

```
pvm_bcast()
pvm_scatter()
pvm_gather()
pvm_reduce()
```

operate with defined group of processes.

Process joins named group by calling **`pvm_joingroup()`**

Multicast operation, **`pvm_mcast()`**, is not a group operation.

Sample PVM program.

Master

```
#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;
  mytid=pvm_mytid();/*Enroll in PVM */

/* Start Slave Tasks */
  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]);
    printf(“%d from %d\n”,result[who],who);
```

Slave

```
#include <stdio.h>
#include “pvm3.h”
#define PROC 10
#define NELEM 1000

main() {
  int mytid;
  int tids[PROC];
  int n, me, i, msgtype;
  int x, nproc, master;
  int data[NELEM], sum;
```



```
/* 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]);

/* 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);


/* Get results from Slaves*/
   msgtype = 5;
   for (i=0; i<nproc; i++){
      pvm_recv(-1, msgtype);
      pvm_upkint(&who, 1, 1);
      pvm_upkint(&result[who], 1, 1);
      printf(“%d from %d\n”,result[who],who);
   }

/* Compute global sum */
   for (i=0; i<nproc; i++) tot += result[i];
   printf (“The total is %d.\n\n”, tot);

  pvm_exit(); /* Program finished. Exit PVM */
  return(0);
}
```

Broadcast data

Receive results

```
   mytid = pvm_mytid();

/* Receive data from master */
   msgtype = 0;
   pvm_recv(-1, msgtype);
   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;}

/* 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(PvmDataDefault)
   pvm_pkint(&me, 1, 1);
   pvm_pkint(&sum, 1, 1);
   msgtype = 5;
   master = pvm_parent();
   pvm_send(master, msgtype);

/* Exit PVM */
   pvm_exit();
   return(0);
}
```

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 $n - 1$, where there are n processes.

Other communicators can be established for groups of processes.

Prentice Hall Upper Saddle River New Jersey, USA, ISBN 0-13-671710-1.

Using the SPMD Computational Model

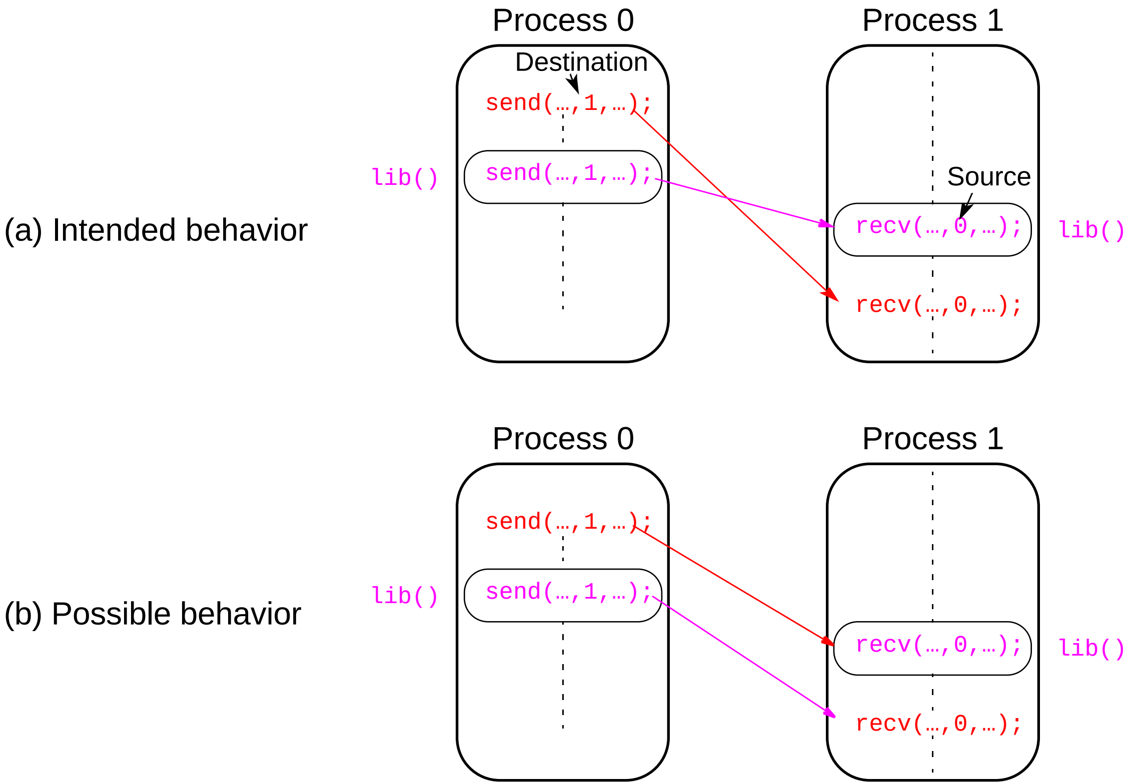
```
main (int argc, char *argv[])
{
MPI_Init(&argc, &argv);
.
.
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/*find process rank */
if (myrank == 0)
    master();
else
    slave();
.
.
MPI_Finalize();
}
```

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



Slides for *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers* by Barry Wilkinson and Michael Allen, Prentice Hall Upper Saddle River New Jersey, USA, ISBN 0-13-671710-1.

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.

Slides for *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers* by Barry Wilkinson and Michael Allen, Prentice Hall Upper Saddle River New Jersey, USA, ISBN 0-13-671710-1.

Point-to-Point Communication

PVM style packing and unpacking data is generally avoided by the use of an MPI datatype being defined.

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)
```

- 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

Parameters of the blocking receive

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

- 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

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.

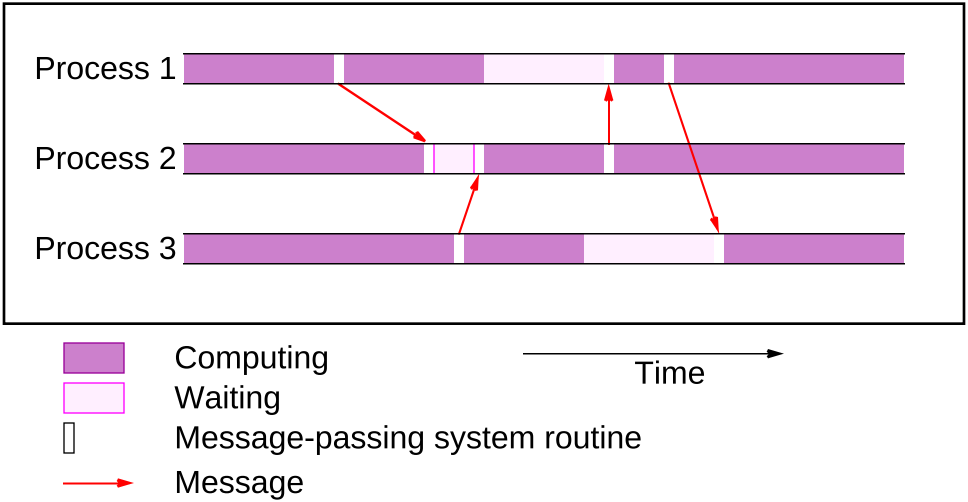
Sample MPI program.

```
#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();
}
```


Debugging and Evaluating Parallel Programs

Visualization Tools

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



Slides for *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers* by Barry Wilkinson and Michael Allen, Prentice Hall Upper Saddle River New Jersey, USA, ISBN 0-13-671710-1.

PVM has a visualization tool called XPVM.

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).

Home Page

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

Basic Instructions for Compiling/Executing PVM Programs

Preliminaries

- Set up paths
- Create required directory structure
- Modify makefile to match your source file
- Create a file (hostfile) listing machines to be used (optional)

Details described on home page.

Compiling/executing PVM programs

Convenient to have two command line windows.

To start PVM:

At one command line:

    pvm

returning a pvm prompt (>)

To compile PVM programs

At another command line in pvm3/src/:

    aimk file

To execute PVM program

At same command line in pvm3/bin/?/ (where ? is name of OS)

    file

To terminate pvm

At 1st command line (>):

    quit

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

Intentionally blank