Message-Passing Computing

Message Passing Interface (MPI)

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

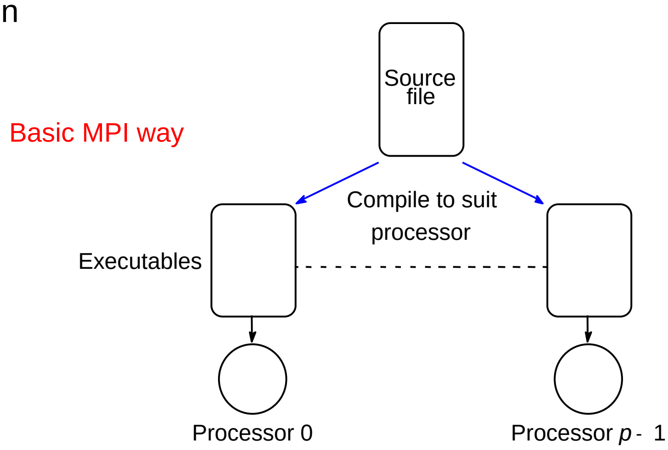
Two primary mechanisms needed:

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

2. A method of sending and receiving messages

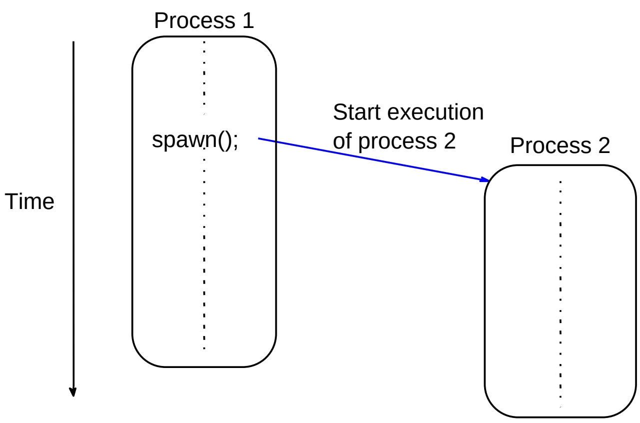
Single Program Multiple Data (SPMD) model

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



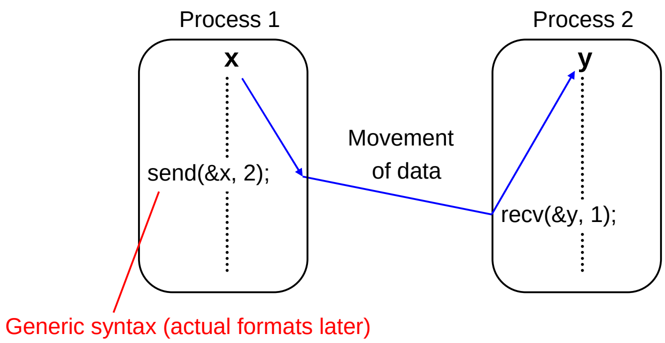
Multiple Program Multiple Data (MPMD) Model

Separate programs for each processor. 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

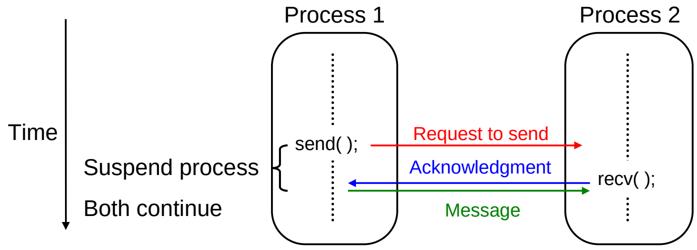
- Waits until complete message can be accepted by the receiving process before sending the message

Synchronous receive routine

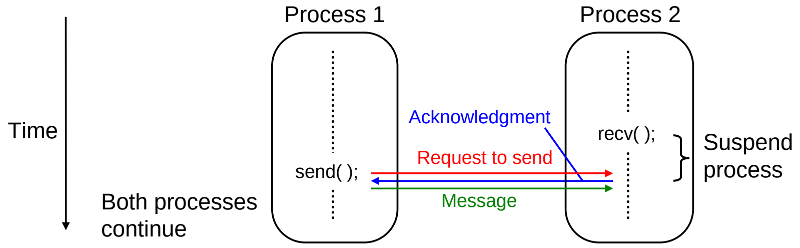
- Waits until the message it is expecting arrives

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

Synchronous send() and recv() 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.

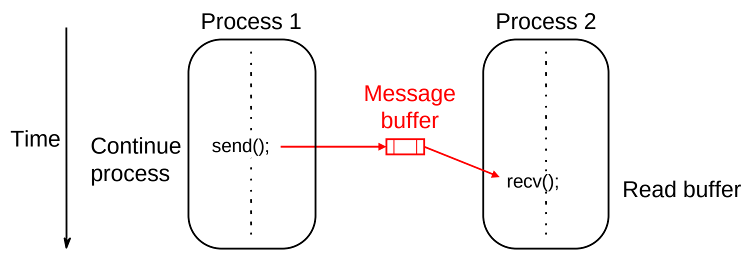
- Non-blocking - return immediately.

  Assumes that data storage 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 routines changing to synchronous routines

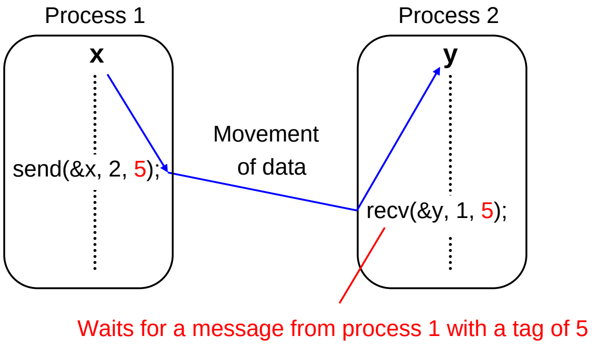
- Once local actions completed and message is safely on its way, sending process can continue with subsequent work.
- Buffers only of finite length and a point could be reached when send routine held up because all available buffer space exhausted.
- Then, send routine will wait until storage becomes 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 with recv(), so that it 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:



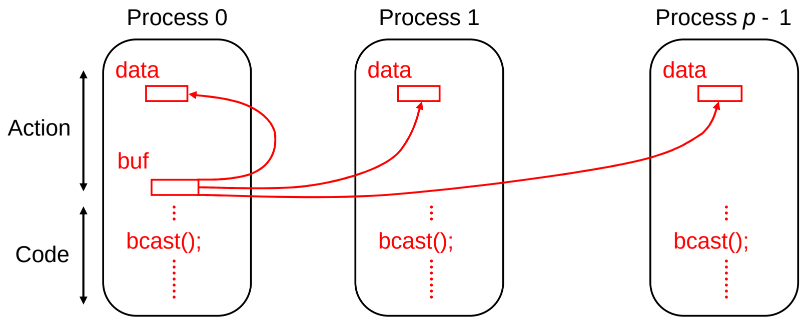
"Group" message passing routines

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 depending on implementation

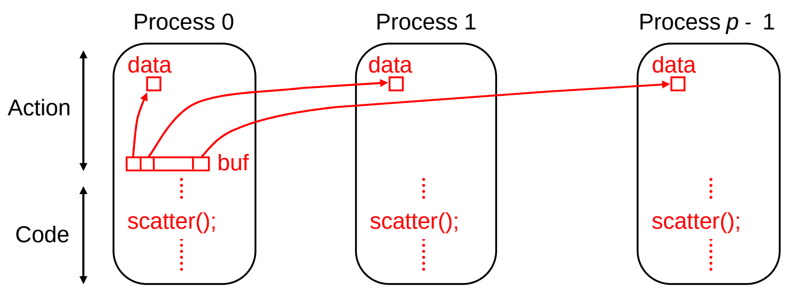
Broadcast

Sending same message to all processes including the root (the sender) process
Multicast - sending same message to a defined group of processes



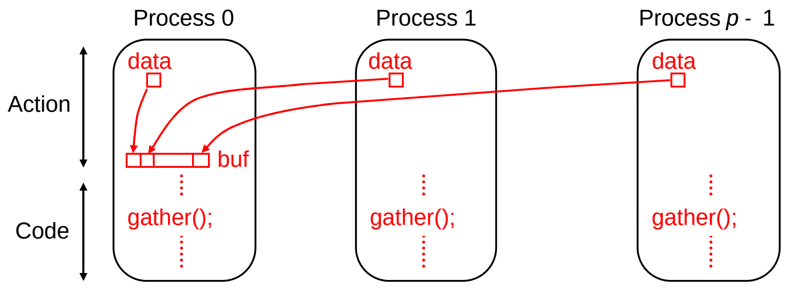
Scatter

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

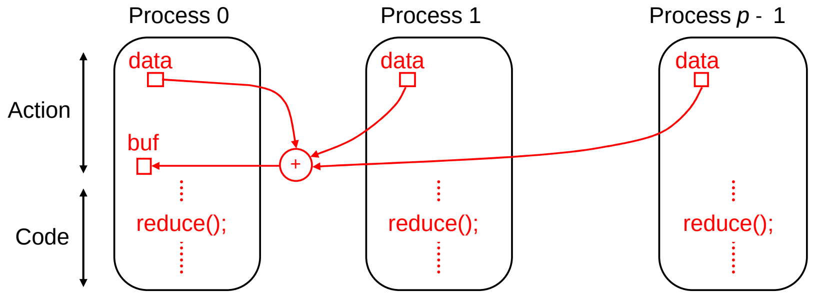
Having one process collect individual values from a set of processes.



Reduce

Gather operation combined with specified arithmetic/logical operation.

Alternative Action: values could be gathered and then added together by root



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

  MPI 1 defined 120+ functions (MPI 2 added more)

  Only need few (about 20) to write programs

  All MPI routines start with the prefix MPI_ (C version)

- Several free implementations exist

MPICH	Argonne National Laboratory
LAM/MPI	Ohio Supercomputing Center

MPI Process Creation and Execution

- Purposely not defined - Will depend upon implementation.

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

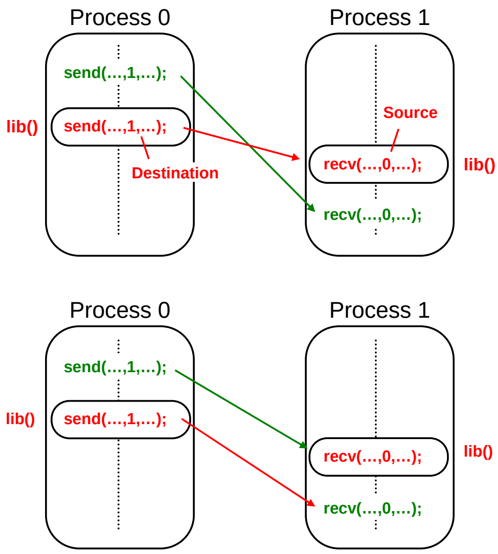
- Originally SPMD model of computation.

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

Unsafe message passing - Example

(a) Intended Behavior

(b) Runtime Behavior
might mix user with
library messages



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.

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 number from 0 to p - 1, with p processes.
- Other communicators can be established for 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 process rank */

        if (myrank == 0)
                master();
        else
                slave();
                .
                .
        MPI_Finalize();
}
```

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

Initializing and Ending MPI

- Initialize MPI Environment
  MPI_Init(&argc, &argv)
  Before calling any MPI function

  argc is the argument count (main function)
  argv is the argument vector (main function)

- Terminate MPI execution environment
  MPI_Finalize( )

MPI Point-to-Point Communication

- Uses send and receive routines with message tags as well as communicator
- Wild card message tags available

MPI Blocking Routines

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

Parameters of Blocking Send

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

void *	buf	Address of send buffer
int	count	Number of items to send
MPI_Datatype	datatype	Datatype of each item
int	dest	Rank of destination process
int	tag	Message tag
MPI_Comm	comm	Communicator

Parameters of Blocking Receive

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

void *	buf	Address of receive buffer (loaded)
int	count	Max number of items to receive
MPI_Datatype	datatype	Datatype of each item
int	src	Rank of source process
int	tag	Message tag
MPI_Comm	comm	Communicator
MPI_Status *	status	Status after operation (returned)

Wildcards and Status in MPI_Recv

- MPI_ANY_SOURCE matches any source
- MPI_ANY_TAG matches any tag
- Status is a return value

`status -> MPI_SOURCE`	rank of source
`status -> MPI_TAG`	tag of message
`status -> MPI_ERROR`	potential errors

Example

To send an integer x from process 0 to process 1

```
int myrank;
int tag = 0;
int x;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */

if (myrank == 0) {
   /* input some value for x */
   MPI_Send(&x, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(&x, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, status);
   . . .
}
```

Some Predefined MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double

Process Rank and Group Size

MPI_Comm_rank(comm, rank)

MPI_Comm_size(comm, size)

MPI_Comm_rank	returns rank of calling process
MPI_Comm_size	returns size of group within comm

MPI_Comm	comm	communicator
int *	rank	process rank (returned)
int *	size	size of group (returned)

MPI Non-Blocking 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.
- The 'I' in 'Isend' and 'Irecv' means Immediate

Non-Blocking Routine Formats

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

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

void *	buf	Address of buffer
int	count	number of items to send/receive
MPI_Datatype	datatype	Datatype of each item
int	dest/src	Rank of destination/source process
int	tag	Message tag
MPI_Comm	comm	Communicator
MPI_Request *	request	Request handle (returned)

Completion Detection

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

MPI_Wait(request, status)
Wait until operation completes and then return

MPI_Test(request, flag, status)
Test for completion of a non-blocking operation

MPI_Request *	request	request handle
MPI_Status *	status	same as return status of MPI_Recv()
int *	flag	true if operation completed (returned)

Example

Send an integer x from process 0 to process 1 and allow process 0 to continue

```
int myrank;
int tag = 0;
int x;
MPI_Request req;
MPI_Status status;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    MPI_Isend(&x,1,MPI_INT,1,tag,MPI_COMM_WORLD,&req);
    compute();
    MPI_Wait(&req, &status);
} else if (myrank == 1) {
    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 – cont'd

- Each of the four modes can be applied to both blocking and non-blocking send routines.

MPI_Send()	MPI_Isend()	Standard send
MPI_Bsend()	MPI_Ibsend()	Buffered send
MPI_Ssend()	MPI_Issend()	Synchronous send
MPI_Rsend()	MPI_Irsend()	Ready send

- Only the standard mode is available for the blocking and non-blocking receive routines.
- Any type of send routine can be used with any type of receive routine.

Collective Communication

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

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

Broadcast message from root to all processes in comm and to itself

MPI_Scatter (sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm)

Scatter a buffer from root in parts to group of processes

MPI_Gather (sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm)

Gather values for group of processes including root

Example on MPI_Gather

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

```
int *buf;    /* buffer space is
allocated dynamically */

int data[10];    /* data sent
from all processes */
```

Collective Communication – Cont’d

MPI_Alltoall (sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm)

Send data from all processes to all processes

MPI_Reduce (sendbuf, recvbuf, count, datatype
operation, root, comm)

Combines values on all processes to a single value at root

MPI_Scan (sendbuf, recvbuf, count, datatype,
operation, comm)

Computes the partial reductions of data on a collection of processes

MPI_Barrier(comm)

Block process until all processes have called it

Sample MPI program

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#define MAXSIZE 100000
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]);
   }
```

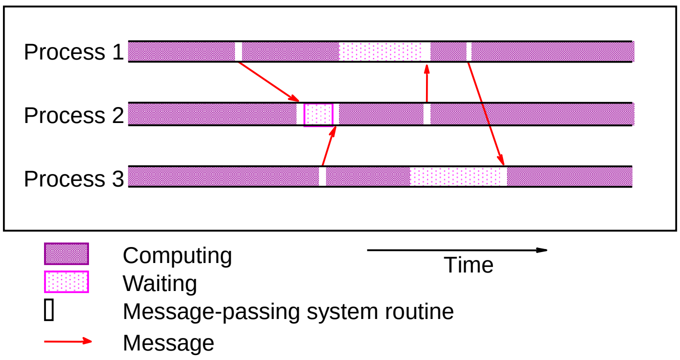
Sample MPI program – cont'd

Can you guess what this program is doing?

```
    /* 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();
}
```


Visualization Tools

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



Evaluating Programs Empirically
Measuring Execution Time

To measure the execution time between two points in the code, we might have a construction such as

```
double t1 = MPI_Wtime();   /* start timer */
              .
              .
double t2 = MPI_Wtime();   /* stop timer */
              .
elapsed_time = t2 - t1;    /* in seconds */
printf("Elapsed time = %5.2f seconds", elapsed_time);
```

MPI provides the routine **`MPI_Wtime()`**

Returns time in seconds since an arbitrary time in past

Step by Step Instructions

for Compiling and Executing

Programs Using MPICH

(Class Demo)