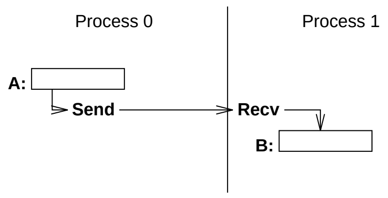


Figure 1: MPI messages.

1 Sending and Receiving messages

Questions:

- To whom is data sent?
- What is sent?
- How does the receiver identify it?

1.1 Current Message-Passing

Message = data + envelope

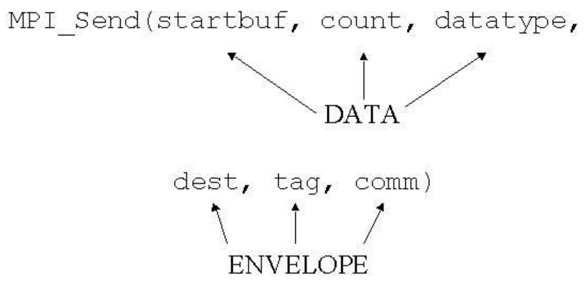
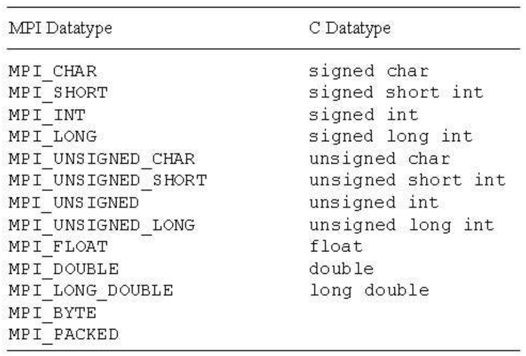


Figure 2: Data+Envelope.

- <u>*MPI Data; Arguments*</u>
  - **startbuf** (starting location of data)
  - **count** (number of elements)
    - receive count $\geq$ send count

- **datatype** (basic or derived)
  - receiver datatype = send datatype (unless MPI_PACKED)
  - elementary (all C and FORTRAN types)
  - derived datatype
    - mixed datatypes
    - contiguous arrays of datatypes
    - strided blocks of datatypes
    - indexed array of blocks of datatypes
    - general structure
  - Specifications of elementary datatypes allows heterogeneous communication.
  - MPI basic datatypes for C:



MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Figure 3: MPI basic datatypes for C.

- *MPI Envelope; Arguments*
  - **destination or source**
    - rank in a communicator
    - receive = sender or MPI_ANY_SOURCE
  - **tag**
    - integer chosen by programmer
    - receive = sender or MPI_ANY_TAG (wild cards allowed)
  - **communicator**
    - defines communication "space"
    - group + context

    - receive = send
  - Collective operations typically operated on all processes.
  - MPI provides groups of processes
    - initial ”all” group.
    - group management routines (build, delete groups).
  - All communication (not just collective operations) takes place in groups.
  - A context partitions the communication space
    - A message sent in one context cannot be received in another context.
    - Contexts are managed by the system.
  - A group and a context are combined in a communicator.
  - Source/destination in send/receive operations refer to rank in group associated with a given communicator.

All of these specifications are a good match to hardware, easy to understand, but too inflexible.

1.2 The Buffer

Sending and receiving only a contiguous array of bytes. Specified in MPI by *starting address*, *datatype*, and *count*

- hides the real data structure from hardware which might be able to handle it directly.
- requires pre-packing dispersed data
  - rows of a matrix stored columnwise.
  - general collections of structures.
- prevents communications between machines with different representations (even lengths) for same data type

1.3 Sample Program using Library Calls

Sub1 and *Sub2* are from different libraries.

```
Sub1();
Sub2();
```

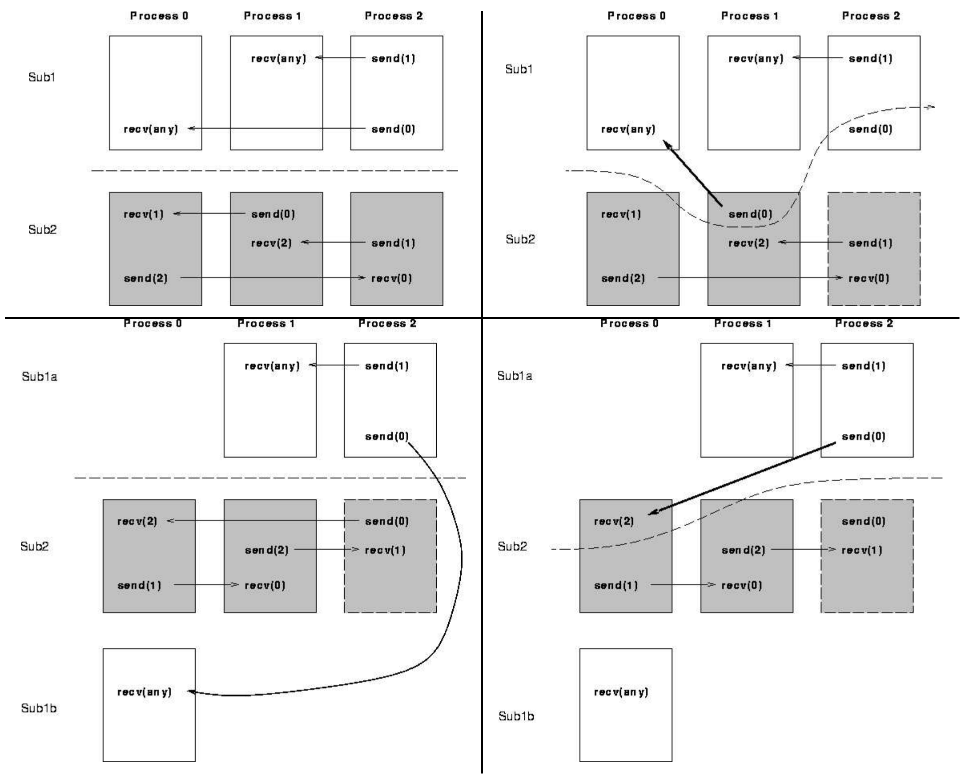


Figure 4: **left-top**; Correct Execution of Library Calls, **right-top**; Incorrect Execution of Library Calls, **left-bottom**; Correct Execution of Library Calls with Pending Communcication, **right-bottom**; Incorrect Execution of Library Calls with Pending Communication.

Sub1a and *Sub1b* are from the same library

```
Sub1a();
Sub2();
Sub1b();
```

1.4 MPI Basic Send/Receive

Thus the basic (blocking) send has become:

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

and the receive:

```
   MPI_Recv(start, count, datatype, source, tag,
                 comm, status)
```

The source, tag, and count of the message actually received can be retrieved from `status`.

```
MPI_Status status;
MPI_Recv( ..., &status );
... status.MPI_TAG;
... status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &count );
```

MPI_Get_count may be used to determine how much data of a particular type was received.

Two simple collective operations:

```
   MPI_Bcast(start, count, datatype, root, comm)
   MPI_Reduce(start, result, count, datatype,
              operation, root, comm)
```

1.5 Exercise/Example: Translate this Fortran code to C

```
      program main
      include 'mpif.h'

      integer rank, size, to, from, tag, count, i, ierr
      integer src, dest
      integer st_source, st_tag, st_count
      integer status(MPI_STATUS_SIZE)
      double precision data(100)

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
      print *, 'Process ', rank, ' of ', size, ' is alive'
      dest = size - 1
      src = 0
C
      if (rank .eq. src) then
         to     = dest
         count  = 10
         tag    = 2001
         do 10 i=1, 10
 10         data(i) = i
         call MPI_SEND( data, count, MPI_DOUBLE_PRECISION, to,
     +                  tag, MPI_COMM_WORLD, ierr )
      else if (rank .eq. dest) then
         tag   = MPI_ANY_TAG
         count = 10
         from  = MPI_ANY_SOURCE
         call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from,
     +                 tag, MPI_COMM_WORLD, status, ierr )
         call MPI_GET_COUNT( status, MPI_DOUBLE_PRECISION,
     +                       st_count, ierr )
         st_source = status(MPI_SOURCE)
         st_tag    = status(MPI_TAG)
C
         print *, 'Status info: source = ', st_source,
     +            ' tag = ', st_tag, ' count = ', st_count
         print *, rank, ' received', (data(i),i=1,10)
      endif

      call MPI_FINALIZE( ierr )
      end
```

1.6 Computation of PI as an example

This example evaluates π by numerically evaluating the integral

$$\int_0^1 \frac{1}{1+x^2} dx = \frac{\pi}{4}$$

- The master process reads number of intervals from standard input, this number is then broadcast to the pool of processes.
- Having received the number of intervals, each process evaluates the total area of **n/pool_size** rectangles under the curve
- The contributions to the total area under the curve are collected from participating processes by the master process, which at the same time adds them up, and prints the result on standard output.

This program computes PI (with a very simple method) but does not use **MPI_Send** and **MPI_Recv**. Instead, it uses *collective* operations to send data to and from all of the running processes.

- The routine **MPI_Bcast** sends data from one process to all others.

- The routine **MPI_Reduce** combines data from all processes (by adding them in this case), and returning the result to a single process.

```
#include "mpi.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  while (!done)
  {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);

    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
}
```

1.7 Exercise

- Experiment with send/receive.
- Run program for PI. Write new versions that replace the calls to **MPI_Bcast** and **MPI_Reduce** with **MPI_Send** and **MPI_Recv**.