Chapter 2

Message-Passing Computing

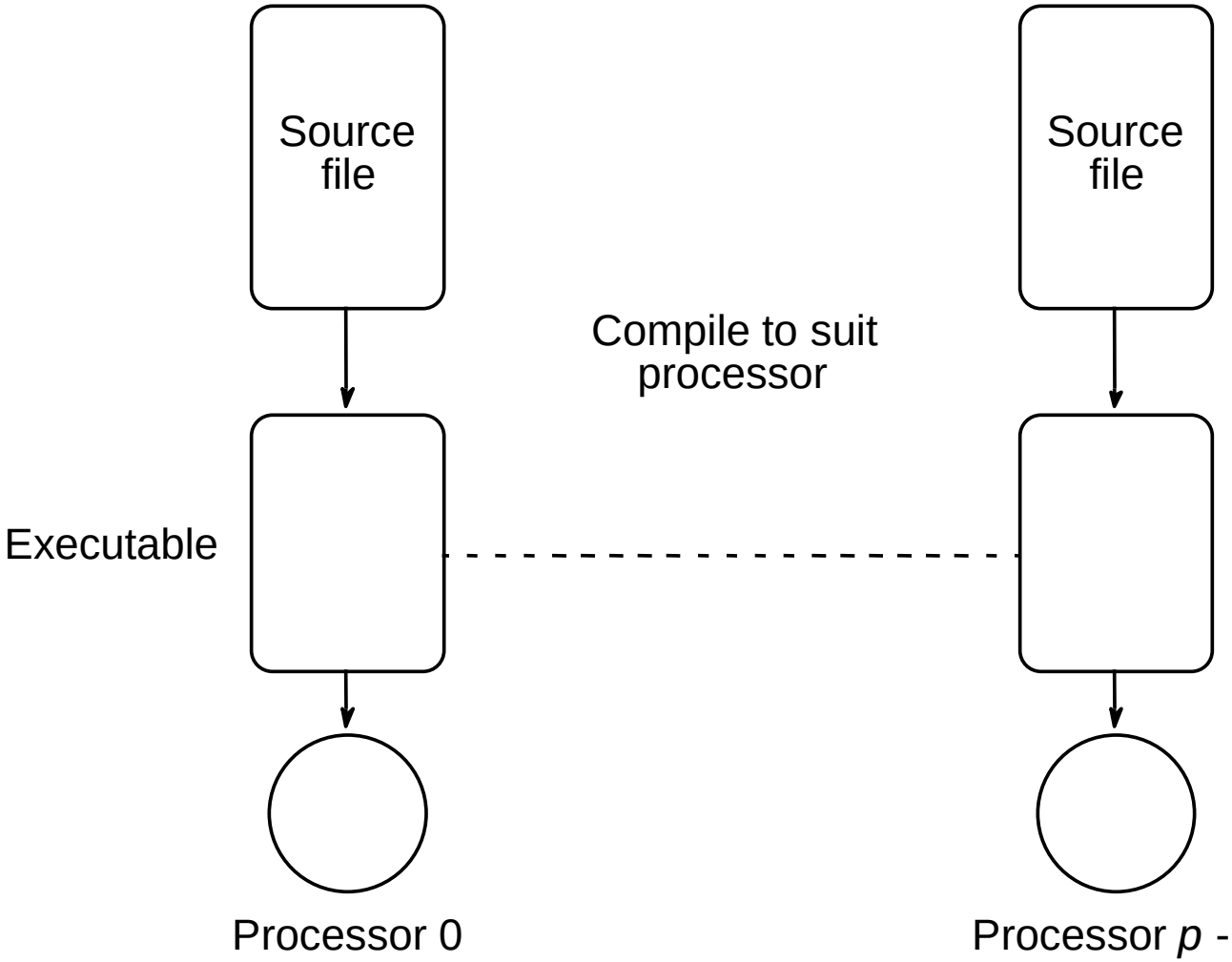
Message-Passing Programming using User-level Message-Passing Libraries

Two primary mechanisms needed:

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

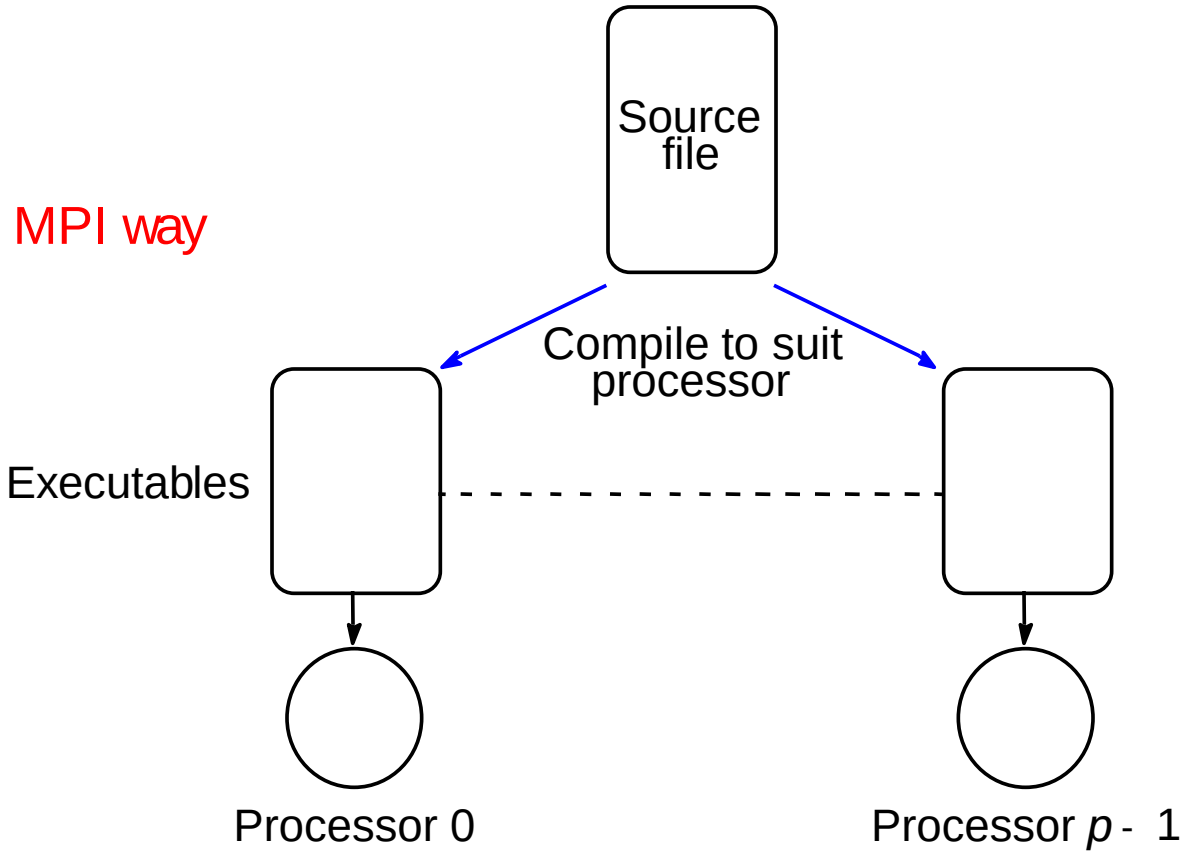
2. A method of sending and receiving messages

Multiple program, multiple data (MPMD) model



Single Program Multiple Data (SPMD) model

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



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 process, respectively.

MPI Dot product code

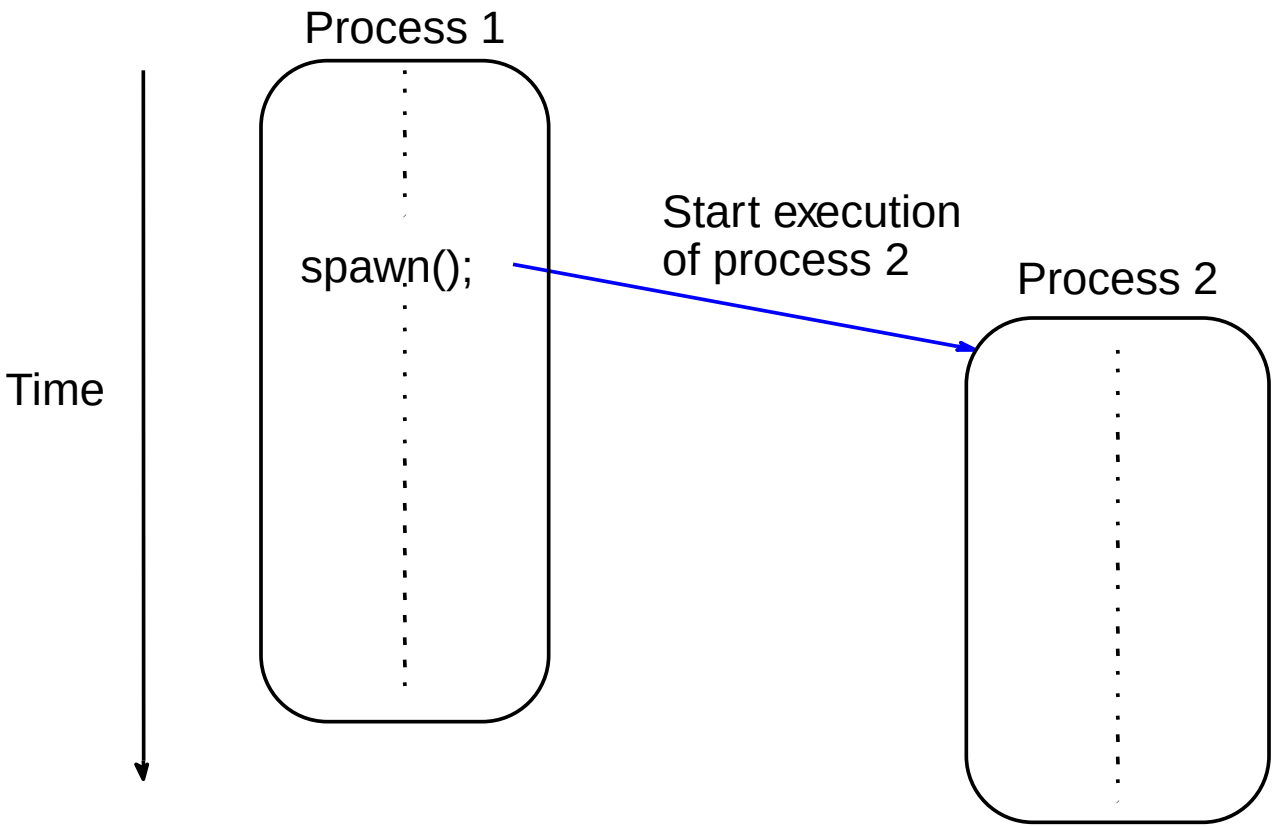
```
int main(argc,argv)
int argc;    char *argv[];
{double sum, sum_local;  double a[256], b[256];
 int n; numprocs, myid, my_first, my_last;
 n = 256;

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
 my_first = myid*n/numprocs;
 my_last  = (myid+1) * n/numprocs;

 for (i =0; i < n; i++) {
    a[i] = i*0.5;    b[i] = i*2.0;          }
 sum = 0.0;
 for (i = my_first; i< my_last; i++) {
    sum_local =sum_local  + a[i]*b[i];  }
 MPI_Allreduce(&sum_local,&sum, 1,MPI_DOUBLE, MPI_SUM,
                        MPI_COMM_WORLD);
 if(myid == 0)printf (" sum= %f",sum);
 }
```

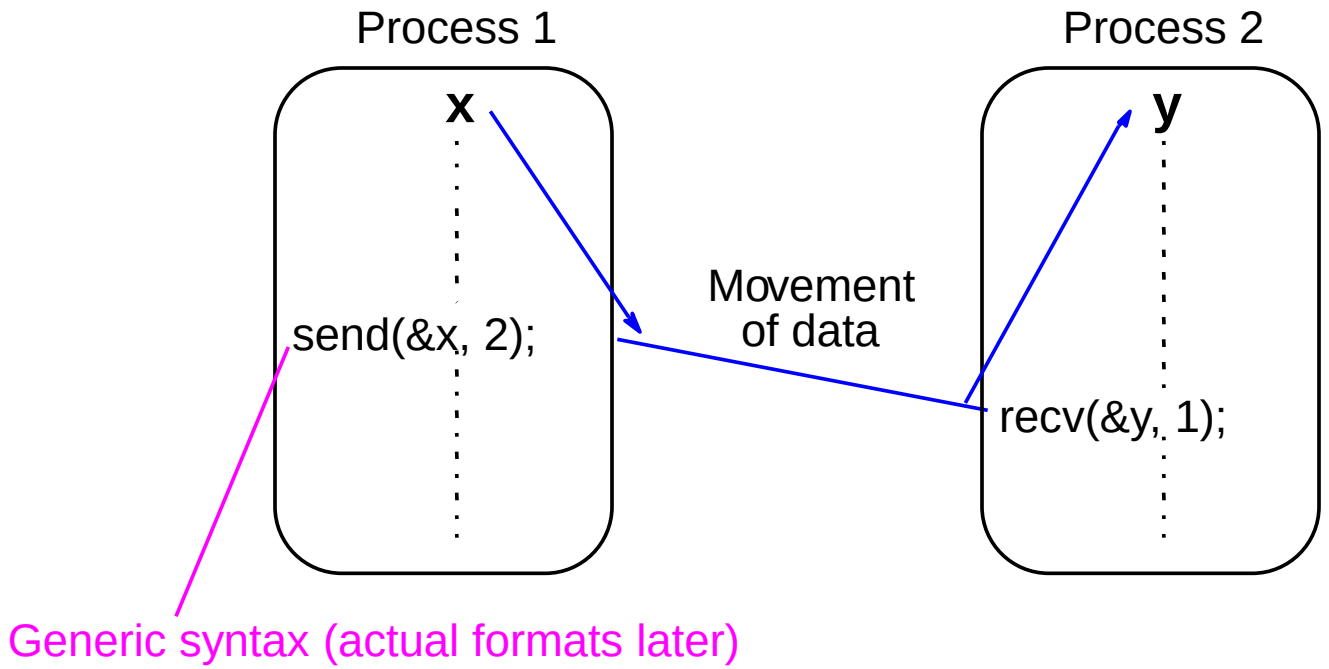
Multiple Program Multiple Data (MPMD) Model

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

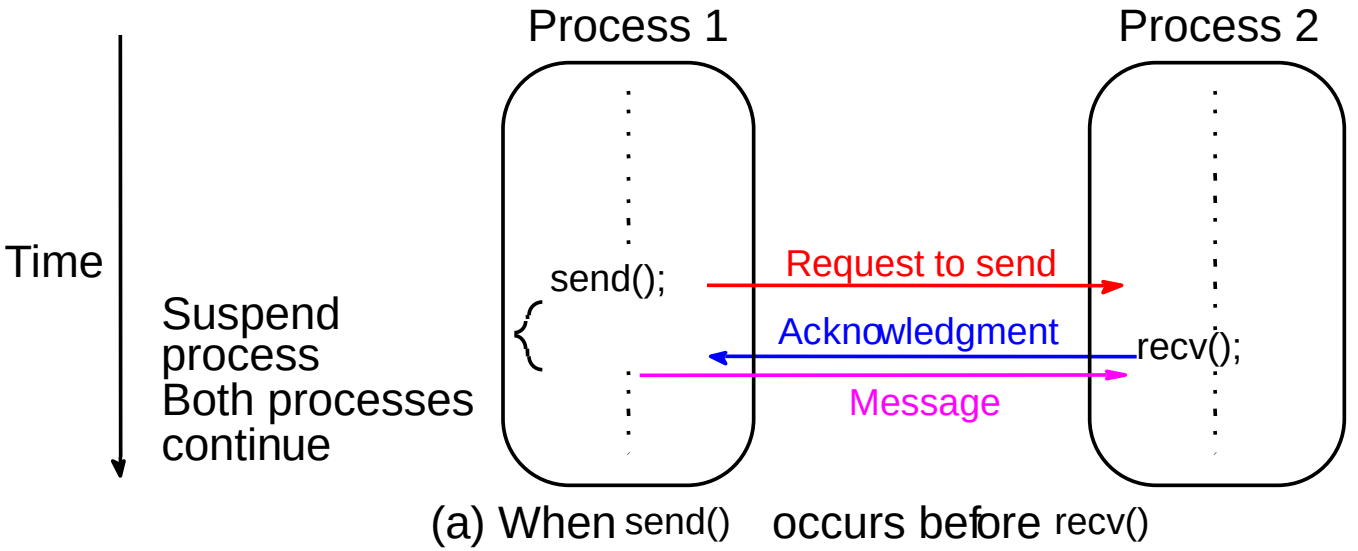
- Waits until complete message can be accepted by the receiving process before sending the message.

Synchronous receive routine

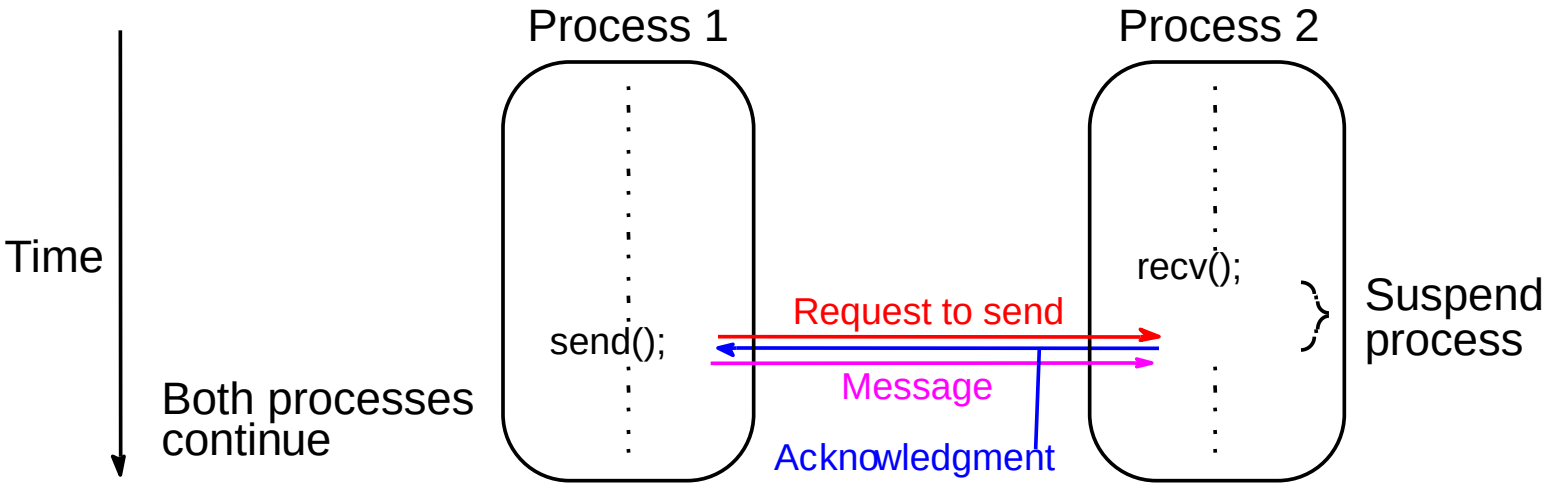
- Waits until the message it is expecting arrives.

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.

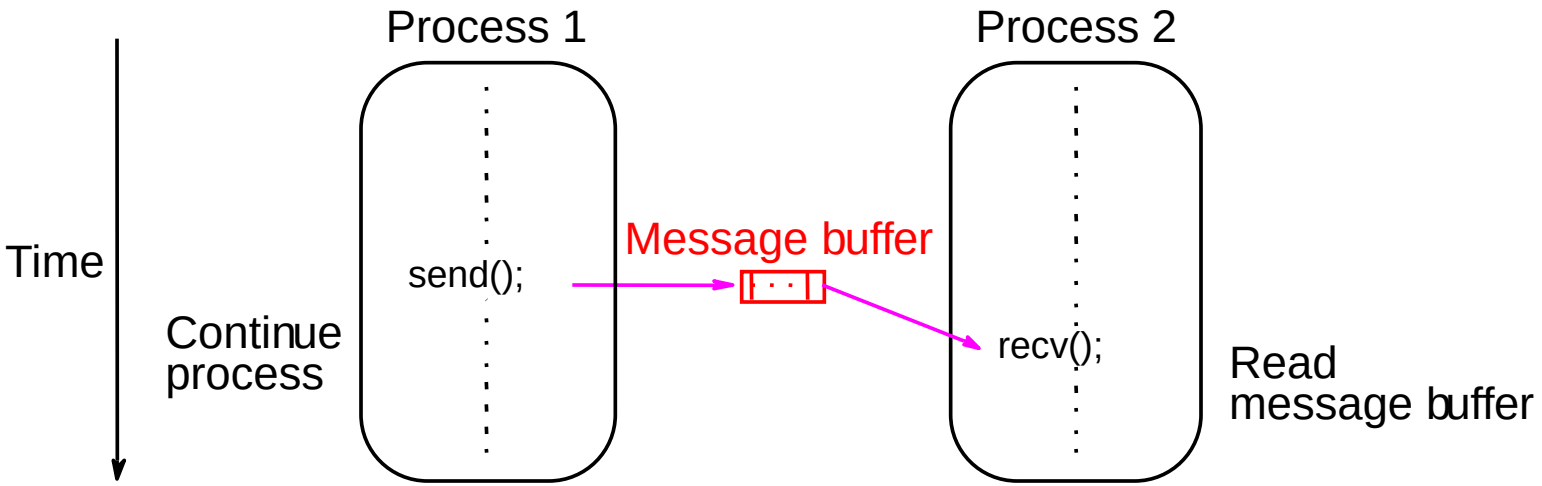
- Non-blocking - return immediately.

  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 (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.

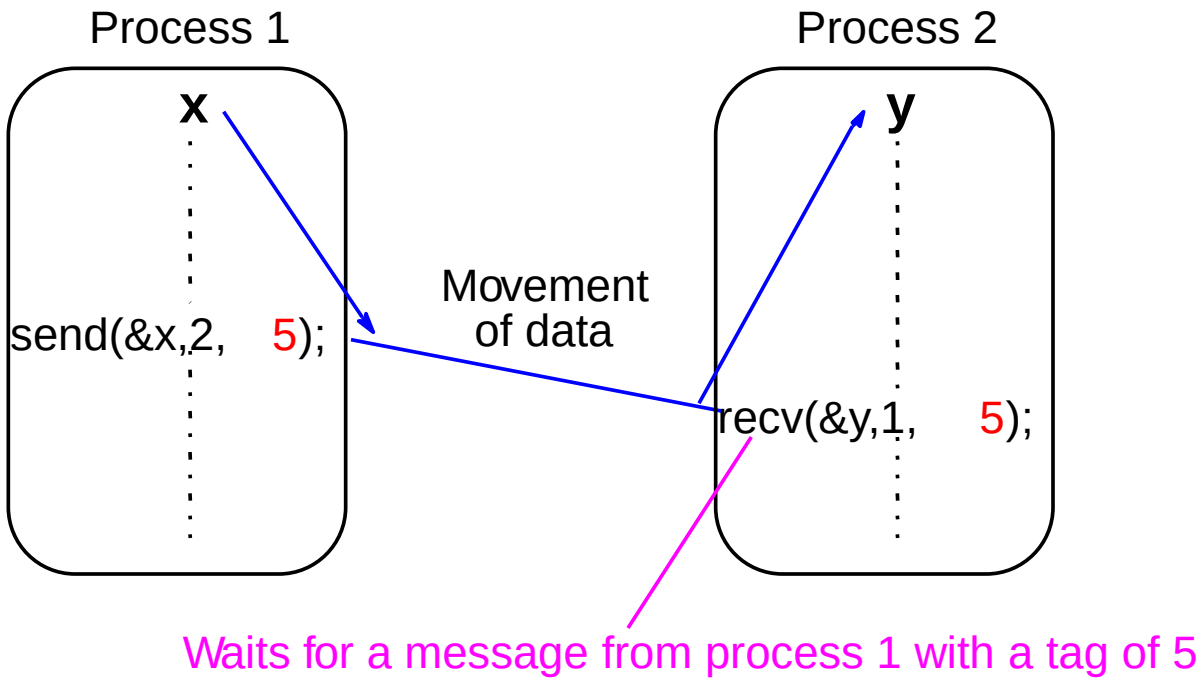
- Then, send routine will wait until storage becomes re-available - i.e then routine behaves as a synchronous routine.

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

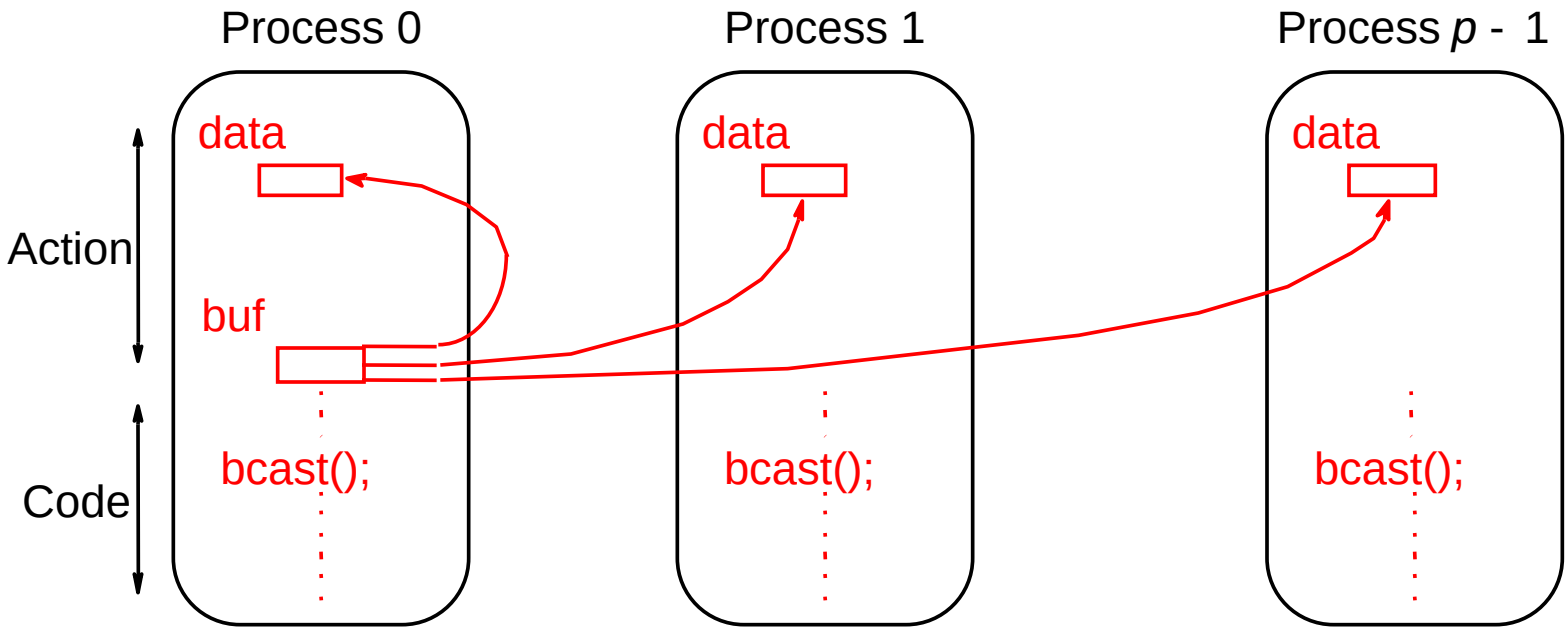
Have routines that send message(s) to a group of processes or receive message(s) from a group of processes

Higher efficiency than separate point-to-point routines although not absolutely necessary.

Broadcast

Sending same message to all processes concerned with problem.

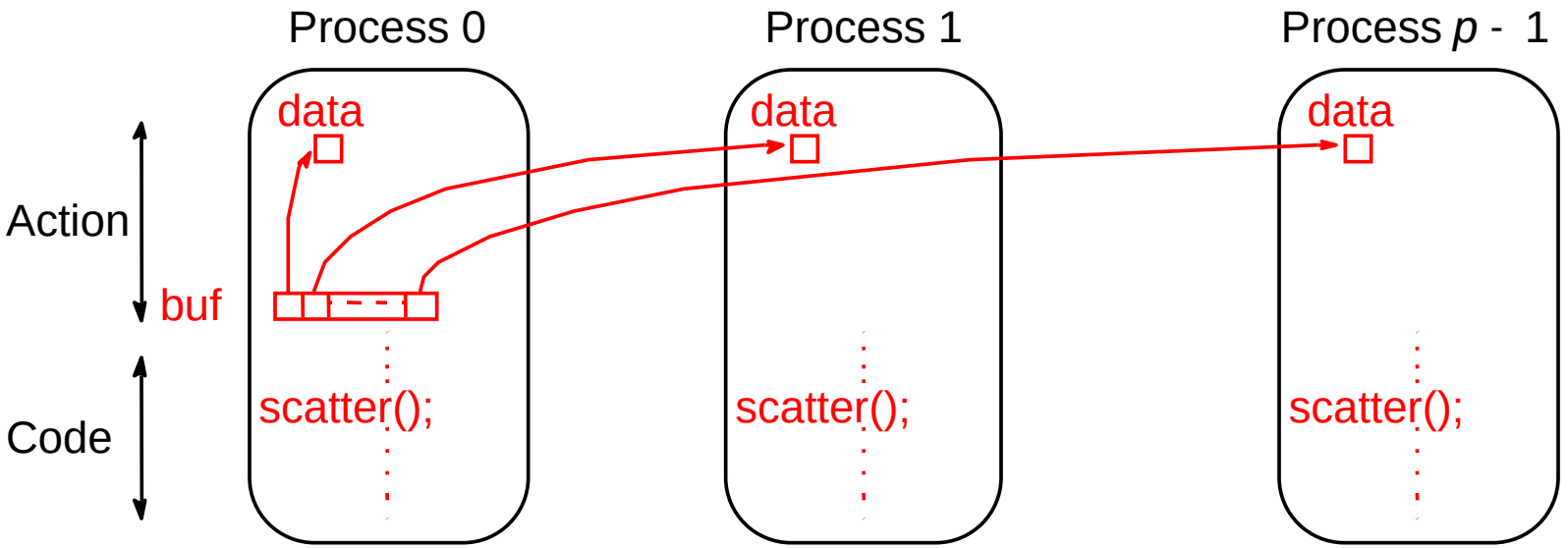
Multicast - sending same message to defined group of processes.



MPI form

Scatter

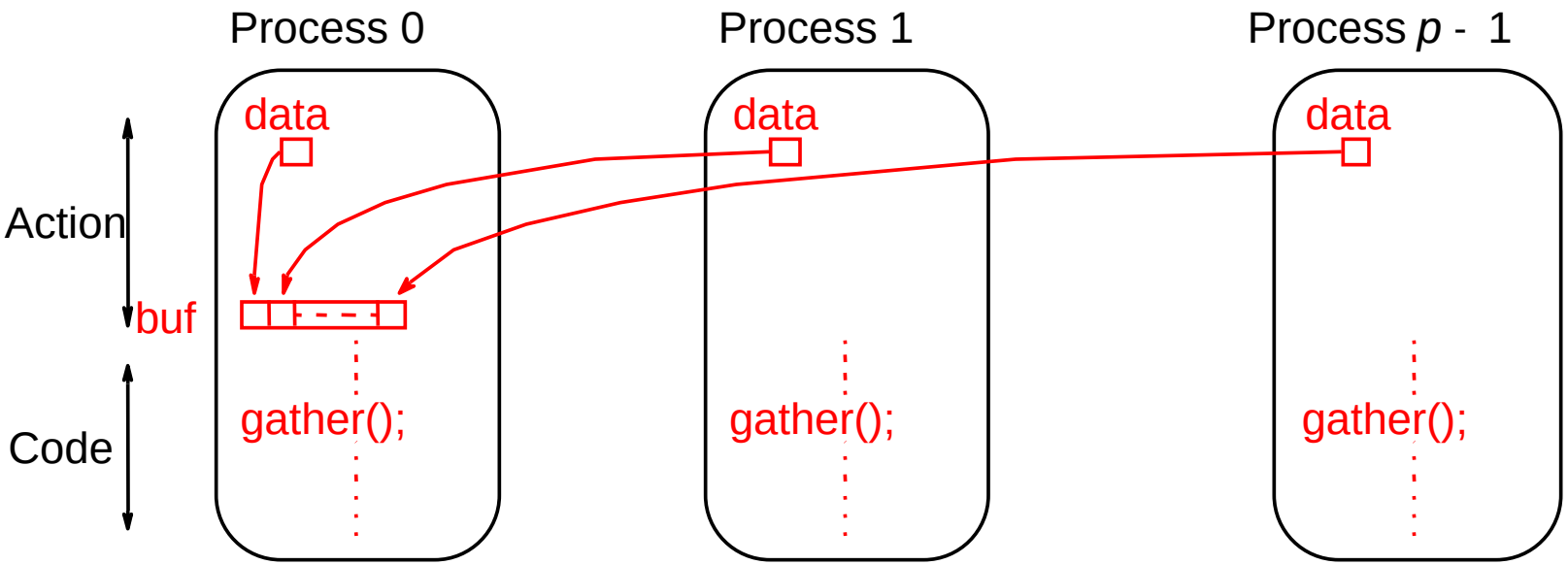
Sending each element of an array in root process to a separate process. Contents of *i*th location of array sent to *i*th process.



MPI form

Gather

Having one process collect individual values from set of processes.

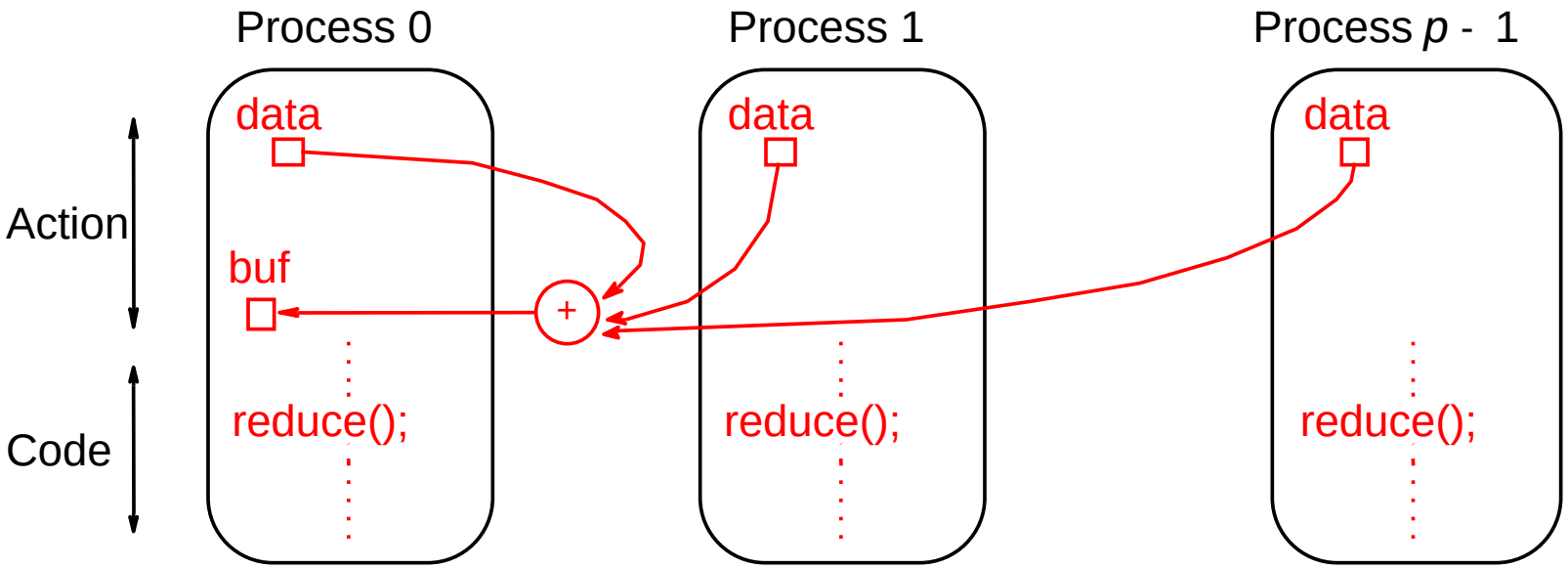


MPI form

Reduce

Gather operation combined with specified arithmetic/logical operation.

Example: Values could be gathered and then added together by root:

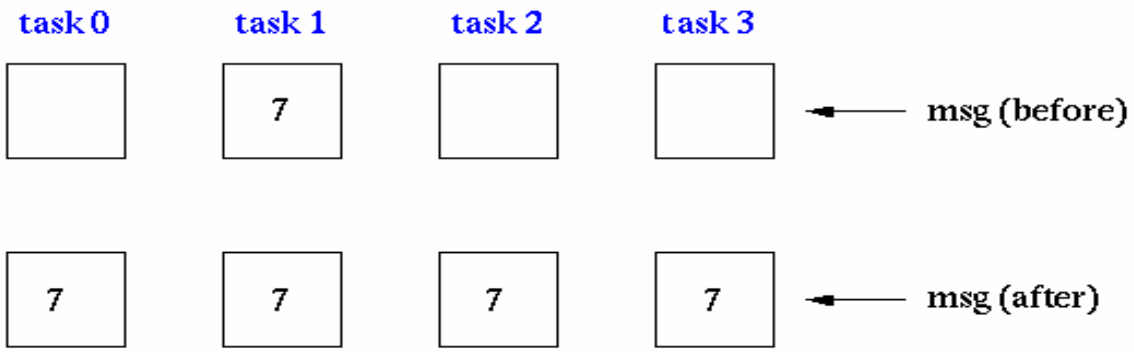


MPI form

MPI_Bcast

Broadcasts a message to all other processes of that group

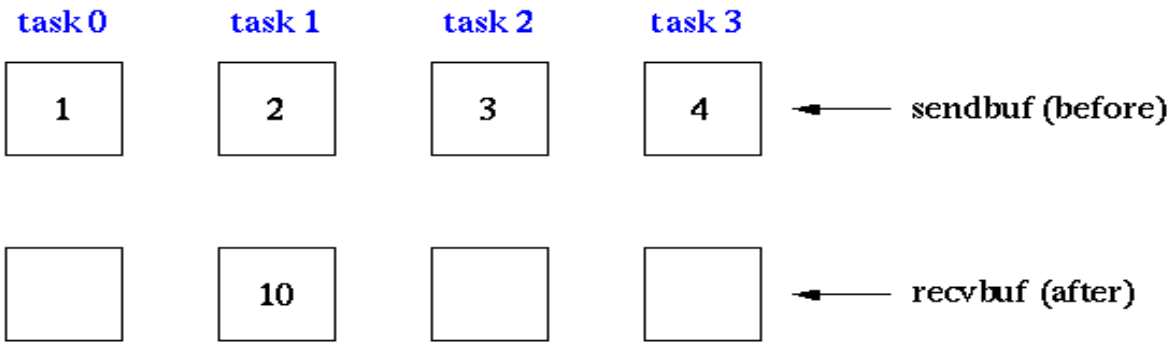
```
count = 1;
source = 1;                    broadcast originates in task 1
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```



MPI_Reduce

Perform and associate reduction operation across all tasks in the group and place the result in one task

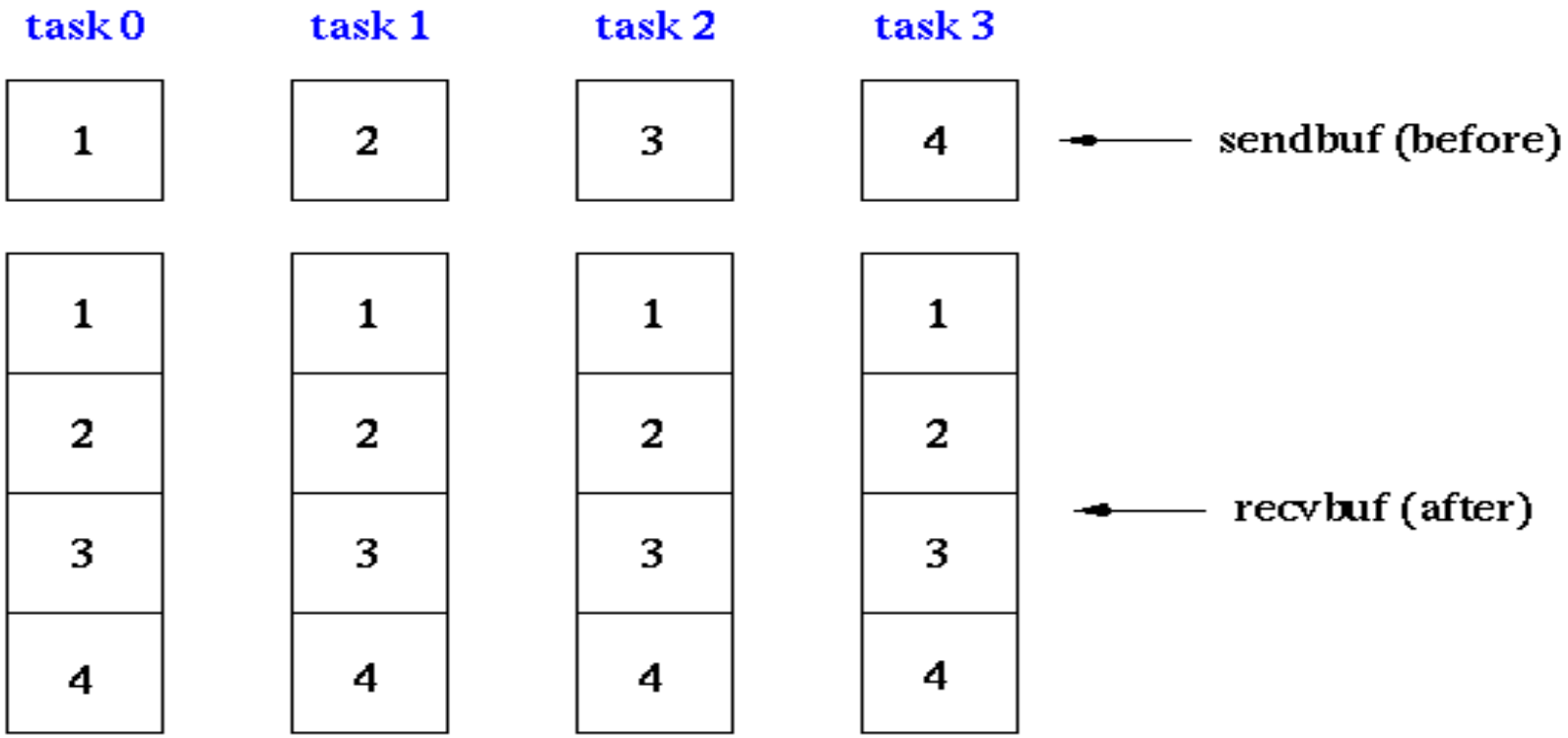
```
count = 1;
dest = 1;                      result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
           dest, MPI_COMM_WORLD);
```



MPI_Allgather

Gathers together values from a group of processes and distributes to all

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT,
              recvbuf, recvcnt, MPI_INT,
              MPI_COMM_WORLD);
```



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.
- Several free implementations exist.
- Examples of Different Implementations
  - MPICH - developed by Argonne Nationa Labs (freeware)
    MPI/LAM - developed by Indiana, OSC, Notre Dame (freeware)
    MPI/Pro - commerical product
  - Apple's X Grid
    OpenMPI - MPI-2 compliant, thread safe
- We use OpenMPI and MPICH have used MPI/LAM

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. Dynamic Process creation in MPI-2

- Originally SPMD model of computation.
- MPMD also possible with static creation - each program to be started together specified. But

SPMD with if (myrank) is equivalent to MPMD

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, with p processes.
- Other communicators can be established for groups of processes.

Using SPMD Computational Model

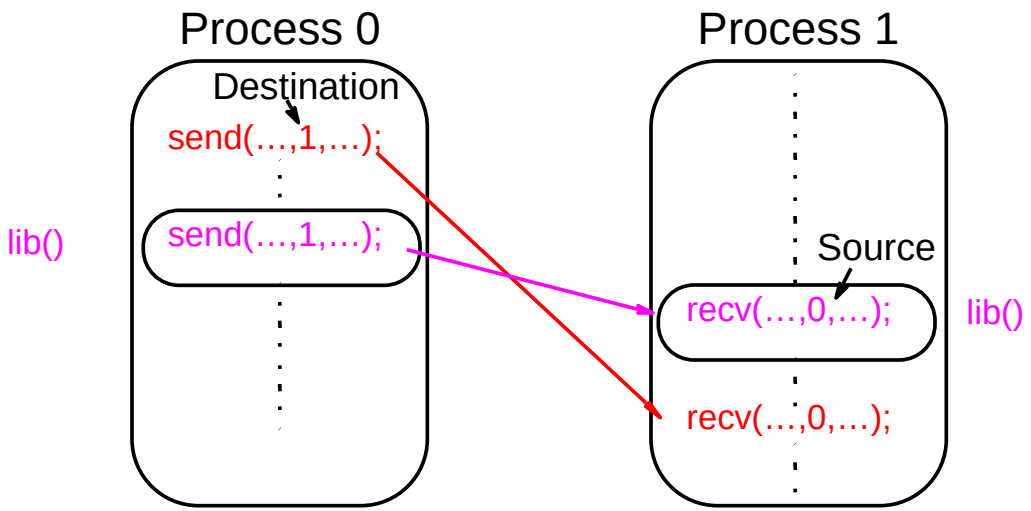
```
main (int argc, char *argv[])
{
   MPI_Init(&argc, &argv);
                .
                .
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank */

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

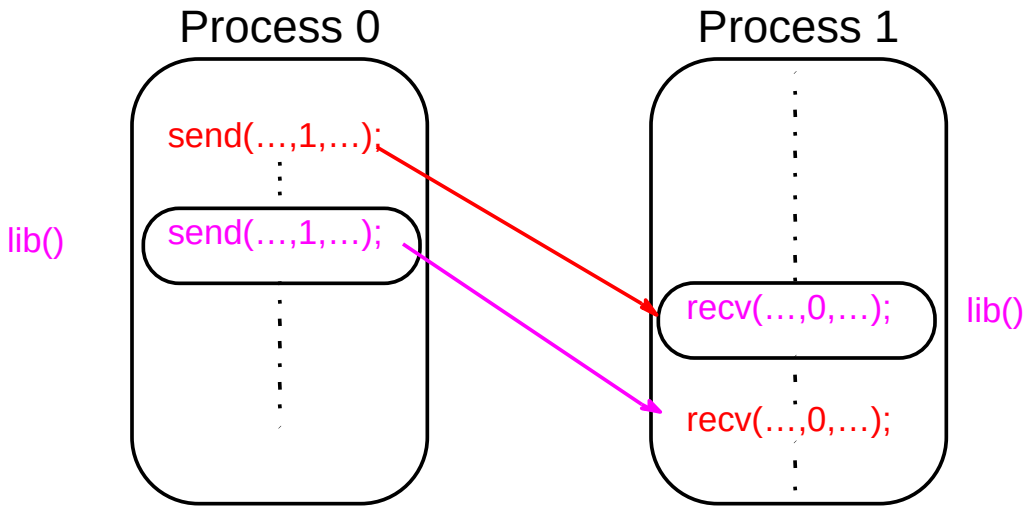
where master() and slave() are to be executed by master process and slave process, respectively.

Unsafe message passing - Example

(a) Intended behavior



(b) Possible behavior



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.
- Many possibilities see e.g. http://static.msi.umn.edu/tutorial/scicomp/general/MPI/communicator.html

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.

MPI Point-to-Point Communication

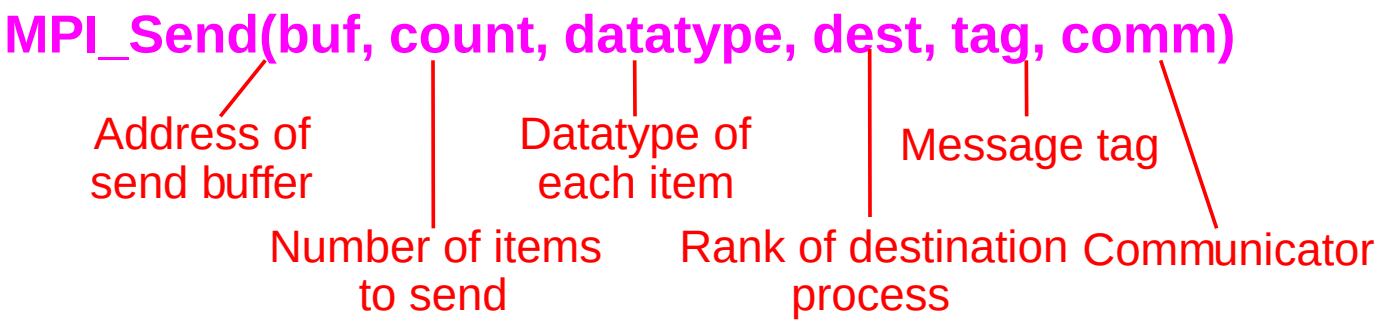
- Uses send and receive routines with message tags (and communicator).
- Wild card message tags available

MPI Blocking Routines

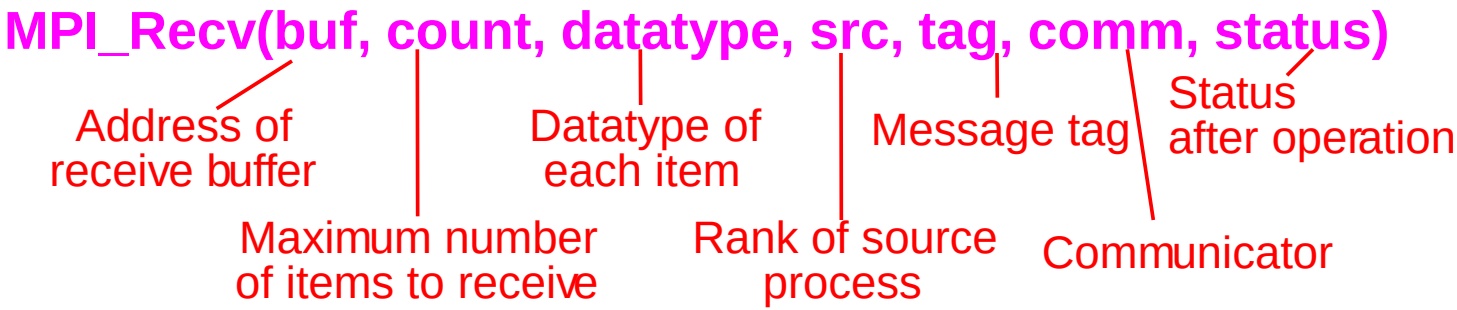
- Return when “locally complete” - when location used to hold message can be used again or altered without affecting message being sent.

- 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



Parameters of blocking receive



In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the structure may contain additional fields. Thus, status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

See http://www.mpi-forum.org/docs/mpi-2.1/mpi21-report-bw/node45.htm

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);
}
```

MPI 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 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 **`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);
}
```

Example Use of Non-Blocking Communication

```
#include <stdio.h>   /* functions sprintf, printf and BUFSIZ defined there */
#include <mpi.h>   /* all MPI functions defined there */
main(argc, argv)
int argc;
char *argv[];
{    int pool_size, my_rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  if (my_rank == 0) {
          MPI_Request request;
          MPI_Status status;
         sprintf (send_buffer, "Dear Task 1,\n\Please do not send any more messages.\n\
          Please send money instead.\n\ \tYours faithfully,\n\ \tTask 0\n\);
          MPI_Isend (send_buffer, strlen(send_buffer) + 1, MPI_CHAR, 1, 77, MPI_COMM_WORLD,
&request);
          printf("hello there user, I've just started this send\n\ and I'm having a good time relaxing.\n");
          MPI_Wait (&request, &status);
          printf("hello there user, it looks like the message has been sent.\n");
          if (request == MPI_REQUEST_NULL) {
                printf("\tthe send request is MPI_REQUEST_NULL now\n");
            } else {                printf("\tthe send request still lingers\n");  }
    }
```

```
else if (my_rank == 1) {
    char recv_buffer[BUFSIZ];
    int my_name_length, count;
    MPI_Request request;
    MPI_Status status;
   MPI_Irecv (recv_buffer, BUFSIZ, MPI_CHAR, 0, 77, MPI_COMM_WORLD,   &request);
    printf("hello there user, I've just started this receive\n\
              and I'm having a good time relaxing.\n");
    MPI_Wait (&request, &status);
    MPI_Get_count (&status, MPI_CHAR, &count);
    printf("hello there user, it looks like %d characters \have just arrived:\n", count );
    printf("%s", recv_buffer);
    if (request == MPI_REQUEST_NULL) {
        printf("\tthe receive request is MPI_REQUEST_NULL now\n");
      } else {     printf("\tthe receive request still lingers\n");
    }
  }
  MPI_Finalize();
}
```

EXAMPLE OF EXECUTION

```
1:hello there user, I've just started this receive
1:and I'm having a good time relaxing.
0:hello there user, I've just started this send
0:and I'm having a good time relaxing.
0:hello there user, it looks like the message has been sent.
0:    the send request is MPI_REQUEST_NULL now
1:hello there user, it looks like 88 characters have just arrived:
1:Dear Task 1,
1:Please do not send any more messages.
1:Please send money instead.
1:    Yours faithfully,
1:    Task 0
1:    the receive request is MPI_REQUEST_NULL now
```

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.
- Also implementation issues e.g. MPI buffer sizes. Message-size > buffer size dictates synchronization for example. Easier if Message-size < buffer size

Collective Communication

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

- **`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 group of processes into process 0, using dynamically allocated memory in root process:

```
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) ;
```

`MPI_Gather()` gathers from all processes, including root.

Barrier routine

- 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]);
     }
     MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD); /* broadcast data */
     x = n/nproc; /* Add my portion Of data */
     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();
}
```

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]);
   }
```

```
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);
    /* broadcast data */
    x = n/nproc; /* Add my portion Of data */
    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();
}
```

MPI-2:

Intentionally, the MPI-1 specification did not address several "difficult" issues. For reasons of expediency, these issues were deferred to a second specification, called MPI-2 in 1997.

MPI-2 was a major revision to MPI-1 adding new functionality and corrections.

Key areas of new functionality in MPI-2:

- **Dynamic Processes** - extensions that remove the static process model of MPI. Provides routines to create new processes after job startup.
- **One-Sided Communications** - provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations.
- **Extended Collective Operations** - allows for the application of collective operations to inter-communicators
- **External Interfaces** - defines routines that allow developers to layer on top of MPI, such as for debuggers and profilers.
- **Additional Language Bindings** - describes C++ bindings and discusses Fortran-90 issues.
- **Parallel I/O** - describes MPI support for parallel I/O.

- **MPI-3:**
- The MPI-3 standard was adopted in 2012, and contains significant extensions to MPI-1 and MPI-2 functionality including:

  - **Nonblocking Collective Operations** - permits tasks in a collective to perform operations without blocking, possibly offering performance improvements.
  - **New One-sided Communication Operations** - to better handle different memory models.
  - **Neighborhood Collectives** - Extends the distributed graph and Cartesian process topologies with additional communication power.
  - **Fortran 2008 Bindings** - expanded from Fortran90 bindings
  - **MPIT Tool Interface** - This new tool interface allows the MPI implementation to expose certain internal variables, counters, and other states to the user (most likely performance tools).
  - **Matched Probe** - Fixes an old bug in MPI-2 where one could not probe for messages in a multi-threaded environment.
- **More Information on MPI-2 and MPI-3:**
- MPI Standard documents: **http://www.mpi-forum.org/docs/**

Evaluating Parallel Programs

Sequential execution time, t_s: Estimate by counting computational steps of best sequential algorithm.

Parallel execution time, t_p: In addition to number of computational steps, t_{comp}, need to estimate communication overhead, t_{comm}:

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

Computational Time

Count number of computational steps.

When more than one process executed simultaneously, count computational steps of most complex process. Generally, function of n and p, i.e.

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

Often break down computation time into parts. Then

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

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

Communication Time

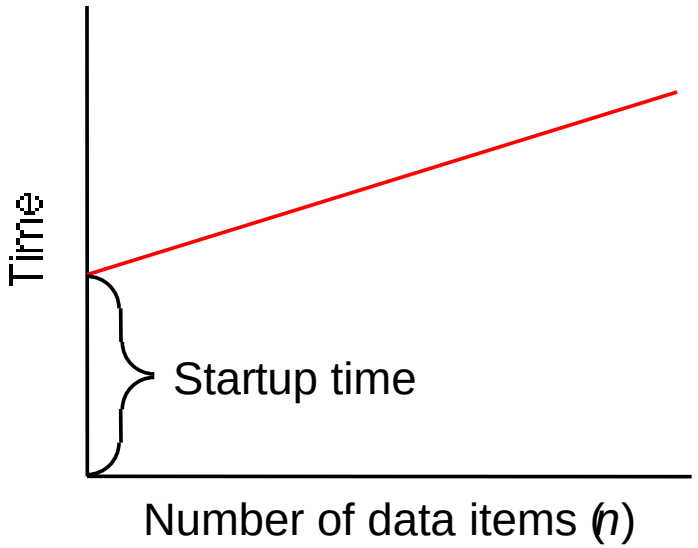
Many factors, including network structure and network contention. As a first approximation, use

$$t_{comm} = t_{startup} + nt_{data}$$

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

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

Idealized Communication Time



In reality the line is not so straight and the architecture of the machine being used matters

Final communication time, t_{comm}

Summation of communication times of all sequential messages from a process, i.e.

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

Communication patterns of all processes assumed same and take place together so that only one process need be considered.

Both $t_{startup}$ and t_{data}, measured in units of one computational step, so that can add t_{comp} and t_{comm} together to obtain 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

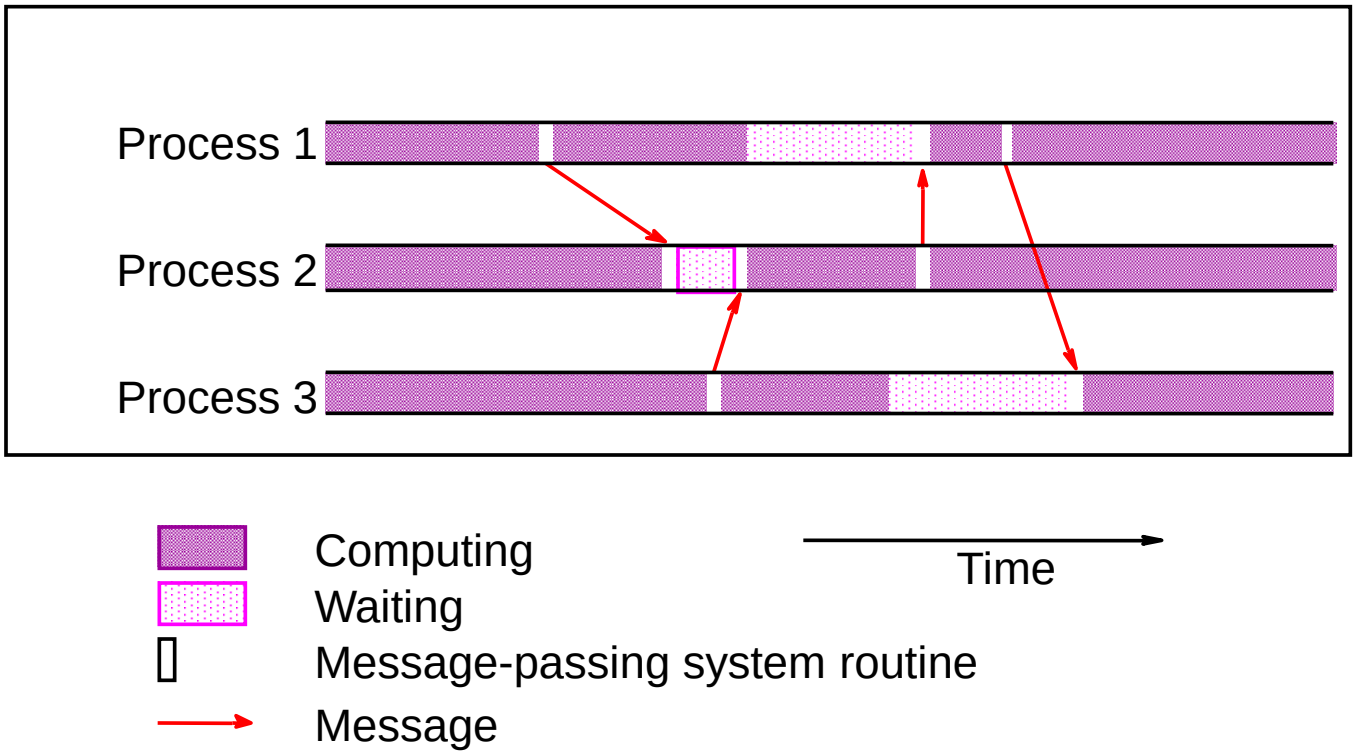
Factors give indication of scalability of parallel solution with increasing number of processors and problem size.

Computation/communication ratio will highlight effect of communication with increasing problem size and system size.

Debugging/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.



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.

Compiling/Executing MPI Programs
Preliminaries

- Set up paths
- Create required directory structure
- Create a file listing machines to be used (required for LAM MPI if using more than one computer)

“hostfile” or “machines” file

For some implementations of MPI (e.g. LAM MPI) before starting MPI for the first time, need to create a “hostfile.”

Only necessary with MPICH if actually using more than one computer (see later).

Compiling/executing (SPMD) LAM MPI programs

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:

for C `mpicc -o prog prog.c`

for C++ `mpiCC -o prog prog.cpp`

To execute MPI program:

```
mpirun -v -np no_procs prog
```

To remove processes for reboot

```
lamclean -v
```

Terminate LAM

```
lamhalt
```

If fails

```
wipe -v lamhost
```

Compiling/executing MPICH programs

For MPICH. At a command line:

To start MPI: Nothing special.

To compile MPI programs:

for C `mpicc -o prog prog.c`

for C++ `mpiCC -o prog prog.cpp`

To execute MPI program:

```
mpirun -v -np no_procs prog
```

verbose mode if desired

A positive integer

Executing MPICH program on multiple computers

Create a file called say “machines” containing the list of machines:

coit-grid01.uncc.edu
coit-grid02.uncc.edu
coit-grid03.uncc.edu
coit-grid04.uncc.edu

mpirun -machinefile machines -np 4 prog

would run prog with four processes.

Each processes would execute on one of the machines in the list. MPI would cycle through the list of machines giving processes to machines.

One can also specify the number of processes on a particular machine by adding that number after the machine name.)

The “MPI standard” command mpiexec is now the replacement for mpirun although mpirun exists.