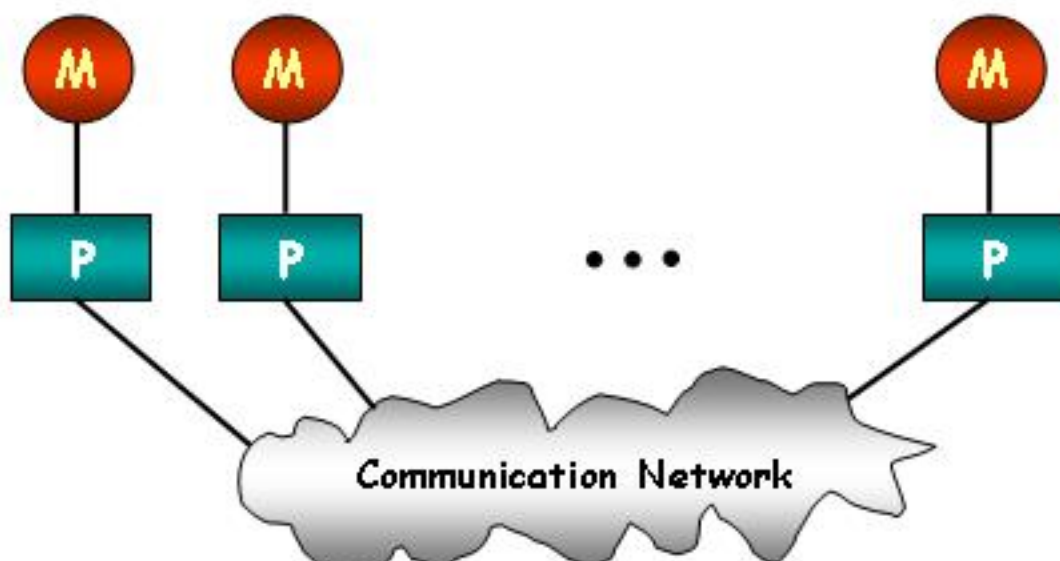


Message-Passing Programming Paradigm

- Many instances of sequential paradigm considered together
- Programmer imagines several processors, each with own memory, and writes a program to run on each processor
- Processes communicate by sending each other messages



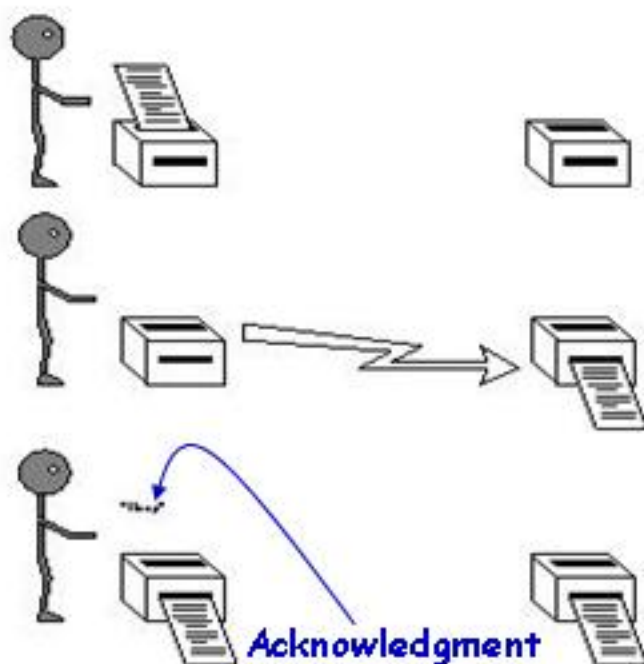
- **Message passing system provides following information to specify the message transfer**
 - **Which process is sending the message**
 - **Where is the data on the sending process**
 - **What kind of data is being sent**
 - **How much data is there**

 - **Which process(es) is/are receiving the message**
 - **Where should the data be left on the receiving process**
 - **How much data is receiving process prepared to accept**

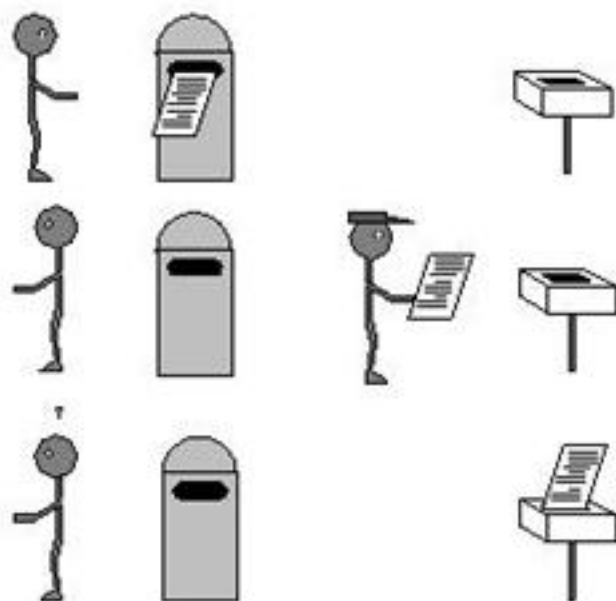
- **Simplest form of message communication**
- **Message is sent from a sending process to a receiving process**
 - **Only these two processes need to know anything about the message**

- **Several variations exist on how sending of a message influences execution of the sub-program**
 - **First common distinction is between *synchronous* and *asynchronous* sends**
 - **Other important distinction is *blocking* and *non-blocking***

- Communication does not complete until the message has been received



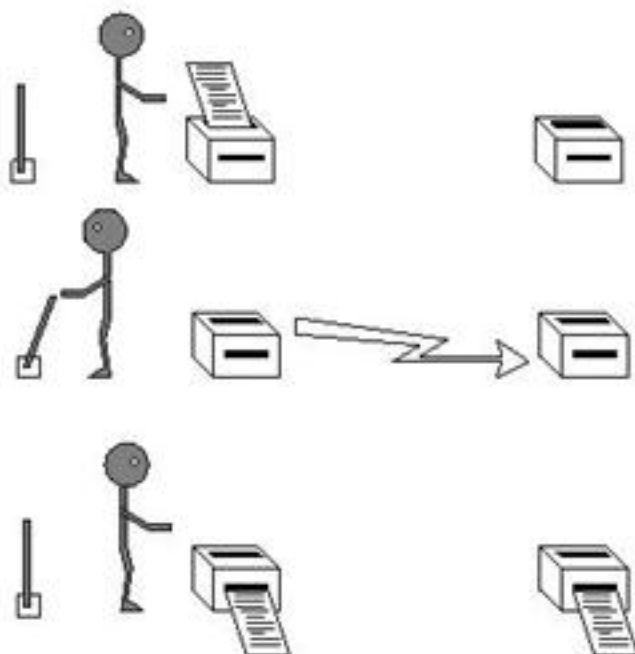
- **Communication completes as soon as message is on its way**
 - *Asynchronous* sends only know when the message has left



- Programs return from the subroutine call when the local transfer operation has completed
 - Though message transfer may not have been completed



- Returns straight away, allows sub-program to perform useful work while waiting for communication to complete
- Sub-program can later test for completion of operation



Message Passing Interface (MPI)

- **A sequential algorithm, in principle, is portable to any architecture supporting sequential paradigm**
 - **However, programmers require more: they want source-code portability**

- **Same should be true for message-passing programs, which forms the main motivation behind MPI**
 - **MPI should provide source-code portability of programs written in C or Fortran across a variety of architectures**
 - **Should enable efficient implementation of parallel programs across range of architectures**

- **MPI is a standard specification for a library of message passing functions**
 - **Contains specifications of functions and macros that can be used in C, FORTRAN, and C++ programs**

- **Developed by the "MPI Forum"**
 - **A Voluntary organization representing industry, government labs and academia, in the form of**
 - **Parallel computer vendors, Library writers, and Application specialist**

- **MPICH2 (V1.1.1p1, Aug 2009)**, is the most popular implementation of MPI
 - Refer <http://www.mcs.anl.gov/research/projects/mpich2/> for download
 - **MPICH2 provides MPI-1 & MPI-2 implementation for clusters, SMPs, and massively parallel processors**
- **CHIMP**, from **Edinburgh Parallel Computing Center**
- **LAM**, from the **Ohio Supercomputing Center**
- **Hardware-specific MPI implementations for**
 - **Cray T3D, IBM SP-2, NEC Cinju, and Fujitsu AP1000**

- **MPI is Large (more than 200 functions)**
 - Many feature requires extensive API
 - Complexity of use not related to number of functions
- **MPI is small (6 functions)**
 - All that's needed to get started are only 6 functions
- **MPI is just right!**
 - Flexibility available when required
 - Can start with small subset

- **Group of processes communicating with each other, through send and receive calls**
- **A communicator argument that defines the group of processes (it's usually `MPI_COMM_WORLD`)**
- **A process can make function calls to**
 - **Find its rank (id)**
 - **Find out the size of its group**
- **A process is made of a typical C/FORTRAN program along with these calls**

```

#include "mpi.h"
#include <stdio.h>

int main( argc, argv)
int argc; char **argv;
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank );
    MPI_Comm_size(MPI_COMM_WORLD, &size );
    /* Your code here */
    printf("Hello world! I'm %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}

```

Header File

Initializing MPI

Communicator

Rank

Size

Exiting MPI

- **To Compile**
 - `mpicc hello.c -o hello`
- **To run with 4 processes**
 - `mpirun -np 4 hello`
- **Output**
 - Hello world! I'm 2 of 4
 - Hello world! I'm 1 of 4
 - Hello world! I'm 3 of 4
 - Hello world! I'm 0 of 4
- **Note - Order of output is not specified by MPI**

<code>MPI_Init</code>	Initializes MPI
<code>MPI_Finalize</code>	Terminates MPI
<code>MPI_Comm_size</code>	Determines the number of processes
<code>MPI_Comm_rank</code>	Determines the label of the calling process
<code>MPI_Send</code>	Sends a message
<code>MPI_Recv</code>	Receives a message

call MPI_Init(ierr)

```
int MPI_Init(int *argc, char **argv)
```

- **Initializes the MPI environment**
- **Called prior to any calls to other MPI routines**

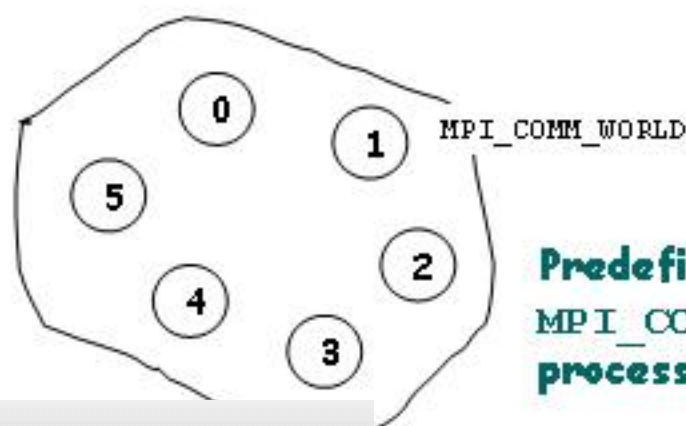
Call MPI_Finalize(ierr)

```
int MPI_Finalize()
```

- **Performs various clean-ups tasks to terminate the MPI environment**
- **Always called at end of the computation**

- **Comprises set of processes that are allowed to communicate with each other**
 - Information about communication domain is stored in variables (called *communicators*) of the type `MPI_Comm`
 - Predefined communicator `MPI_COMM_WORLD` includes *all* processes involved in parallel execution

- **Communicators are used as arguments to all message transfer MPI routines**



**Predefined communicator
MPI_COMM_WORLD for six
processes**

```
call MPI_Comm_size(comm, size, ierr)
```

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- Returns in `size`, the number of processes in communicator `comm`

```
call MPI_Comm_rank(comm, rank, ierr)
```

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- Returns rank of the process in the communicator
- Rank ranges from 0 to (size of the communicator - 1)

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

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

Call `MPI_Recv(buf, count, datatype, source, tag, comm, status, ierr)`

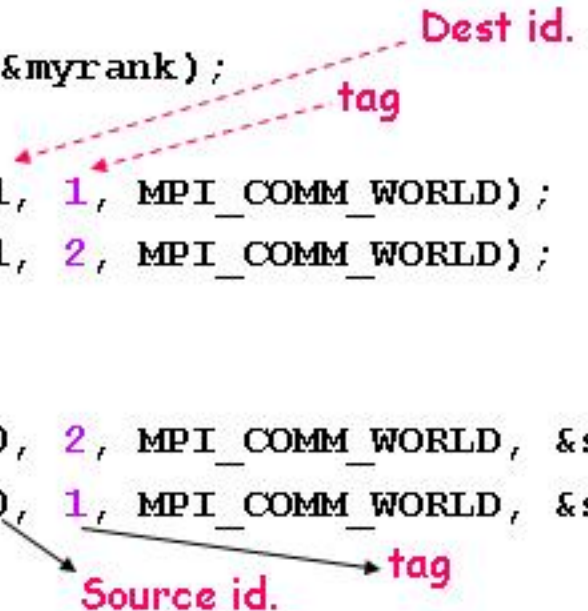
```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

- **The tag (type of message) and source can have wildcard arguments: `MPI_ANY_TAG` & `MPI_ANY_SOURCE`, respectively**
- **If received message is larger than the buffer, routine will return `MPI_ERR_TRUNCATE`**
- **The status is a struct variable giving information about `MPI_Recv` operation**
 - `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`, etc.

MPI_Send when buffered avoids deadlock, else if it blocks until matching receive has been issued, there is a deadlock

```

int in[10], out[10], myrank;
MPI_Status status1, status2;
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank == 0) {
    MPI_Send(in, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(out, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if(myrank == 1) {
    MPI_Recv(out, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, &status1);
    MPI_Recv(in, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &status2);
}
    
```



The diagram illustrates the MPI communication parameters in the code. Red dashed arrows point from the text labels to the corresponding values in the MPI_Send and MPI_Recv calls:

- Dest id.:** Points to the destination rank '1' in the first MPI_Send call and the source rank '0' in the first MPI_Recv call.
- tag:** Points to the tag '1' in the first MPI_Send call and the tag '2' in the first MPI_Recv call.
- Source id.:** Points to the source rank '0' in the second MPI_Recv call.
- tag:** Points to the tag '1' in the second MPI_Recv call.

MPI Datatype	C Datatype	FORTTRAN
<code>MPI_CHAR</code>	<code>signed char</code>	<code>MPI_CHARACTER</code>
<code>MPI_SHORT</code>	<code>signed short int</code>	
<code>MPI_INT</code>	<code>signed int</code>	<code>MPI_INTEGER</code>
<code>MPI_LONG</code>	<code>signed long int</code>	
<code>MPI_UNSIGNED_CHAR</code>	<code>unsigned char</code>	
<code>MPI_UNSIGNED_SHORT</code>	<code>unsigned short int</code>	
<code>MPI_UNSIGNED</code>	<code>unsigned int</code>	
<code>MPI_UNSIGNED_LONG</code>	<code>unsigned long int</code>	
<code>MPI_FLOAT</code>	<code>float</code>	<code>MPI_REAL</code>
<code>MPI_DOUBLE</code>	<code>double</code>	<code>MPI_DOUBLE_PRECISION</code>
<code>MPI_LONG_DOUBLE</code>	<code>long double</code>	
<code>MPI_BYTE</code>		<code>MPI_BYTE</code>
<code>MPI_PACKED</code>		<code>MPI_PACKED</code>

Call `MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierr)`

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

Call `MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierr)`

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Request *request)
```

- **The request object is used by**
 - `MPI_Test(...)` **for querying status of operation**
 - `MPI_Wait(...)` **to wait for its completion**
- **`MPI_Irecv` does not have a status argument**
 - **Instead, the status information is returned by `MPI_Test(...)` and `MPI_Wait(...)`**

Call MPI_Test(request, flag, status, ierr)

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

- Tests for completion of non-blocking send/receive operation
- The flag = true for completion; false, for incomplection

Call MPI_Wait(request, status, ierr)

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- Blocks until non-blocking operation identified by request completes

Replace either send or receive with their non-blocking counterparts; the code would not deadlock

```

int in[10], out[10], myrank;
MPI_Status status;
MPI_Request requests[2];
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank == 0) {
    MPI_Send(in, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(out, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if(myrank == 1) {
    MPI_Irecv(out, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, &requests[0]);
    MPI_Irecv(in, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &requests[1]);
}
    
```

Introduction to MPI

```

/* Single receiver process and N-1 sender processes */
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
    const int tag = 42;                /* Message tag */
    int id, ntasks, source_id, dest_id, err, i, msg[2];
    MPI_Status status;

    err = MPI_Init(&argc, &argv);     /* Initialize MPI */
    if(err != MPI_SUCCESS) {
        printf("MPI initialization failed!\n");
        exit(1);
    }
    /* Get number of tasks and id of this process */
    err = MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    err = MPI_Comm_rank(MPI_COMM_WORLD, &id);

    /* . . . Code for checking if (ntasks >= 2) . . . */

```

```

if(id == 0) { /* Process 0, the receiver, does this */
    for (i=1; i<ntasks; i++) {
        err = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE, tag,
            MPI_COMM_WORLD, &status); /* Receive a message */
        source_id = status.MPI_SOURCE; /* Get id of sender */
        printf("Received message %d of %d from process %d\n",
            msg[0], msg[1], source_id);
    } /* end for */
}
else { /* Processes 1 to N-1 (the senders) do this */
    msg[0] = id; /* Put own identifier in msg. */
    msg[1] = ntasks; /* And total no. of processes */
    dest_id = 0; /* Destination address */
    err = MPI_Send(msg, 2, MPI_INT, dest_id, tag, MPI_COMM_WORLD);
} /* end if */
err = MPI_Finalize(); /* Terminate MPI */
if(id==0)printf("Success!\n");
exit(0);

```

- **To Compile**
 - `mpicc prog1.c -o prog1`
- **To run with 5 processes**
 - `mpirun -np 5 prog1`
- **Output**
 - Received message 2 of 5 from process 2**
 - Received message 3 of 5 from process 3**
 - Received message 1 of 5 from process 1**
 - Received message 4 of 5 from process 4**
 - Success!**
- **Note - Order of output is not specified by MPI**

For sample programs see C-DAC's site:

<http://ctef.cdac.gov.in/hatatast/MPI-1.0/Documents/MPI-hos-contents.html>

- **Introduction to Parallel Computing, 2nd Edition**
 - Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson Education Ltd., 2004 (Low price edition 2004, Rs. 325/-)
[for Introduction (C1, C5:Section 5.2), MPI (C6), OpenMP (C7)]
- **Parallel Computers- Architecture and Programming**
 - V. Rajaraman & C. Siva Ram Murthy, Prentice-Hall (India), 2000
[for Structure of Parallel Computers (C4)]
- **Elements of Parallel Computing**
 - V. Rajaraman, Prentice Hall, 1990
[for Structure of Parallel Computers (Vector Computers only)]
- **Designing and Building Parallel Programs**
 - Ian Foster, Addison-Wesley Inc., 1995
(freely available on <http://www-unix.mcs.anl.gov/dbpp/>)
- **Parallel Computing- Theory and Practice, 2nd edition**
 - Michael J. Quinn, McGraw-Hill, New York, 1994

- **For lecture slides on introduction to parallel computing**
 - http://www.llnl.gov/computing/tutorials/parallel_comp/
- **For lecture slides on MPI**
 - <http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html>
 - <http://www.llnl.gov/computing/tutorials/mpi/> (Lawrence Livermore National Lab)
 - http://www.sdsc.edu/~tkaiser/my_mpi.html
- **For sample MPI programs**
 - <http://ctsf.cdac.org.in/betatest/MPI-1.0/Documents/MPI-hos-contents.html>
- **For OpenMP specifications and reference manual**
 - <http://www.openmp.org>
 - <https://computing.llnl.gov/tutorials/openMP/exercise.html> (OpenMP programs)
- **Others (C-DAC, OPECG 2009)**
 - <http://cdac.ernet.in/html/events/beta-test/opecg/mpi-cpp-opecg-2009/collective-comm-comp-mpi-cpp.html>
 - <http://ctsf.cdac.org.in/betatest/MPI-2.0/Documents/MPI-Information.html>