MPI

Message Passing Interface

References

- For presentation:
  - William Gropp. Ewing Lusk. *An Introduction to MPI Parallel Programming with the Message Passing Interface*. *Argonne National Laboratory*

MPI Sources

- The Standard itself:
  - **at `http://www.mpi-forum.org`**
  - All MPI official releases, in both postscript and HTML

- **Books:**
  - http://mpitutorial.com/recommended-books/

- `http://www.mcs.anl.gov/mpi`

The Message-Passing Model

- A *process* is (traditionally and simply viewed)
  - a program counter and
  - an address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space.
- MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.

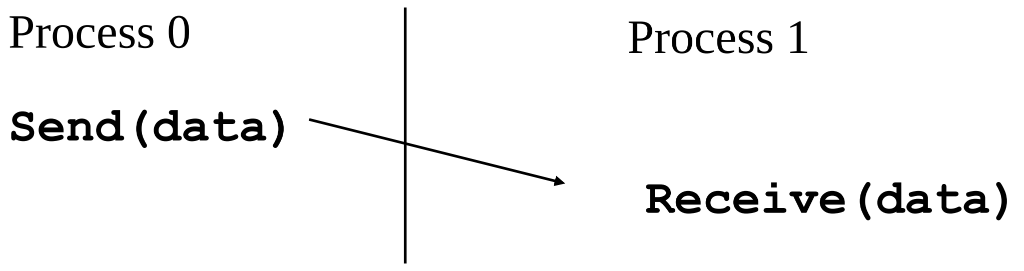
Review: Types of Parallel Computing Models

- **Data Parallel** - the same instructions are carried out simultaneously on multiple data items (**SIMD**)
- **Task Parallel** - different instructions on different data (**MIMD**)
- **SPMD** (single program, multiple data) not synchronized at individual operation level
  - SPMD is equivalent to MIMD since each MIMD program can be made SPMD (similarly for SIMD, but not in practical sense.)

Message passing (and so also MPI) is for MIMD/SPMD parallelism.

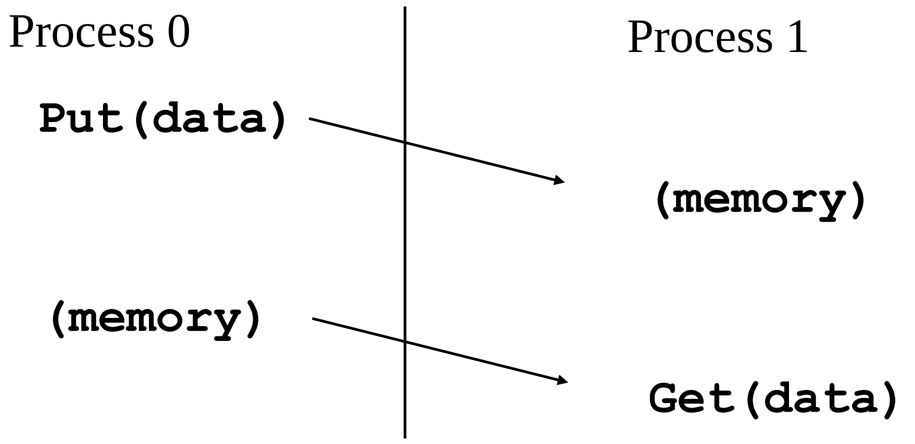
Cooperative Operations for Communication

- The message-passing approach makes the exchange of data *cooperative*.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.



One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
- One-sided operations were included into MPI-2.



What is MPI?

- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

Why Use MPI?

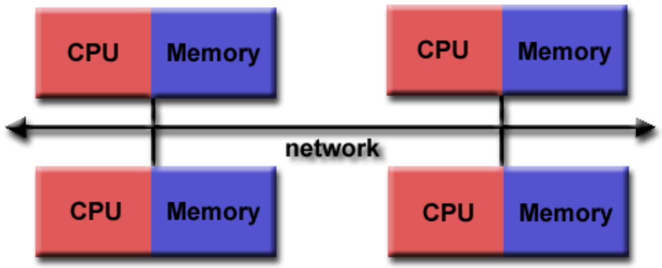
- MPI provides a
  - powerful,
  - efficient, and
  - *portable*

  way to express parallel programs

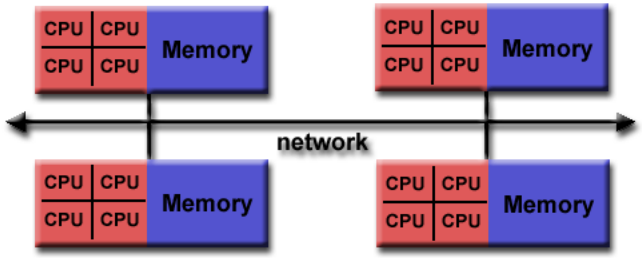
- MPI was explicitly designed to enable libraries…

Architecture compatibility

Initially just for DM



Later also for SM



Platforms:

- Distributed Memory
- Shared Memory
- Hybrid

C, C++ and Fortran bindings

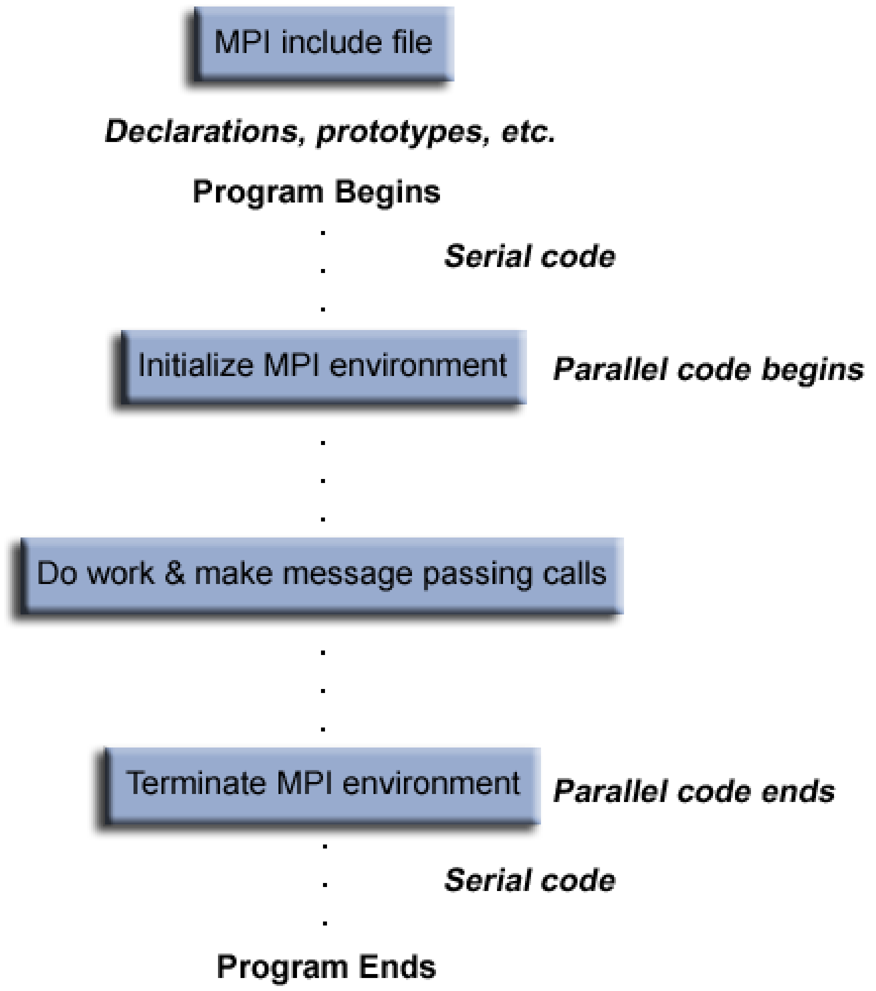
- C and Fortran bindings correspond closely
- In C:
  - mpi.h must be #included
  - MPI functions return error codes or **`MPI_SUCCESS`**
- In Fortran:
  - mpif.h must be included, or use MPI module (MPI-2)
  - All MPI calls are to subroutines, with a place for the return code in the last argument.
- C++ bindings, and Fortran-90 issues, are part of MPI-2.

A Minimal MPI Program (C)

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

Structure of an MPI program


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

Hello World in MPI

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

int main( argc, argv )
int argc;
char **argv;
{
int  namelen, myid, numprocs;
MPI_Init( &argc, &argv );
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
printf( "Process %d / %d  : Hello world\n", myid, numprocs);

MPI_Finalize();
return 0;
}
```

compilation

```
$ mpicc hello.c -o hello
```

execution

```
$ mpirun -np 4 hello
```

```
Process 0 / 4 : Hello world
Process 2 / 4 : Hello world
Process 1 / 4 : Hello world
Process 3 / 4 : Hello world
```

API-> MPI Functions

General format:

rc = MPI_Xxxxx(parameter, ...)

Example:

rc=MPI_Bsend(&buf, count, type, dest, tag, comm)

Error code "rc”:

MPI_SUCCESS ⇔ succes

Error Handling

- By default, an error causes all processes to abort.
- The user can cause routines to return (with an error code) instead.
  - In C++, exceptions are thrown (MPI-2)
- A user can also write and install custom error handlers.
- Libraries might want to handle errors differently from applications.

Running MPI Programs

- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.

- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.

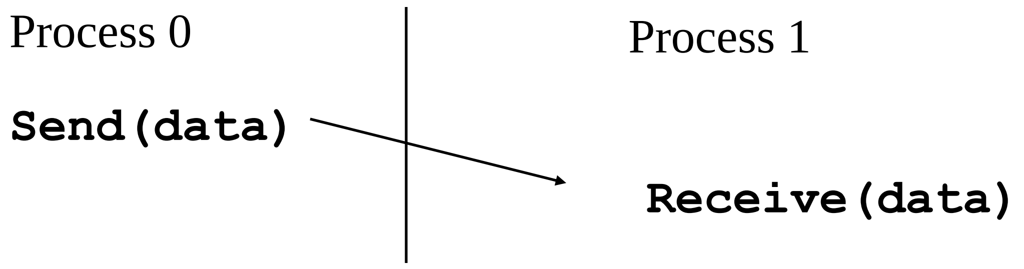
- **`mpiexec <args>`** is part of MPI-2, as a recommendation, but not a requirement
  - **mpiexec**
  - **mpirun**

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

MPI Basic Send/Receive

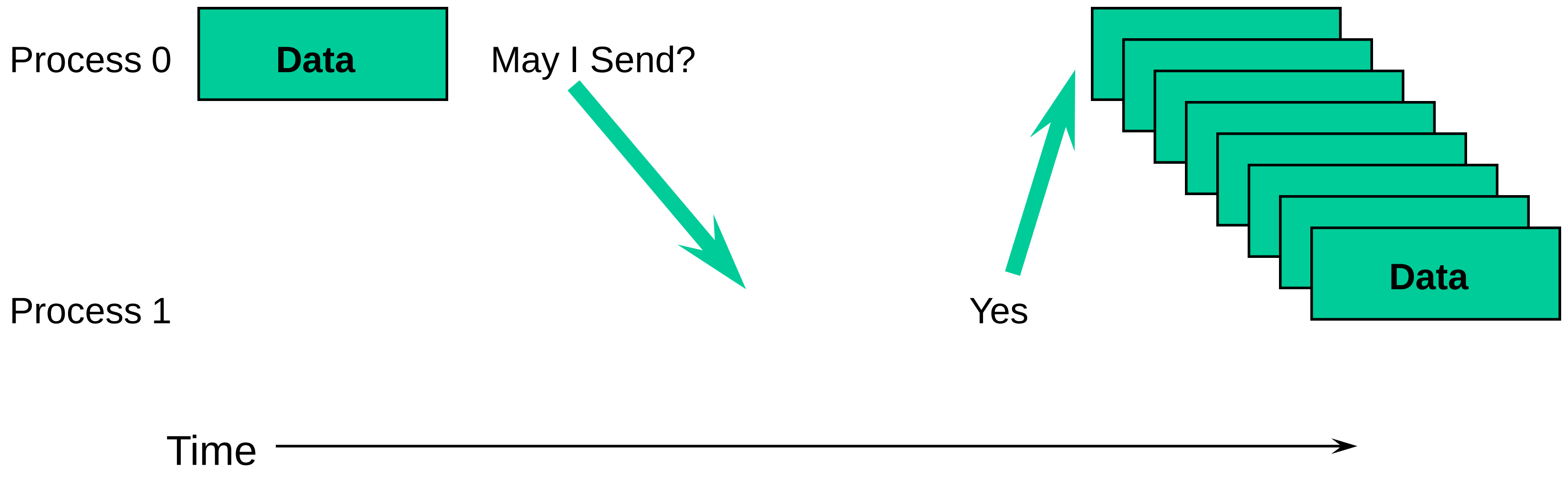
- We need to fill in the details in



- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

What is message passing?

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

MPI Datatypes

- The data in a message to sent or received is described by a triple:
 (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.

MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (on **`source`** and **`tag`**) message is received from the system, and the buffer can be used.
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**.
- **`status`** contains further information
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

Example

```
#include "mpi.h"
#include <stdio.h>
int main(argc,argv)
       int argc;
       char *argv[]; {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
   dest = source = 1;
   rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
&Stat);
}
```

Example (cont)

```
else if (rank == 1){
        dest = source = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &Stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
                         MPI_COMM_WORLD);
}
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d
        \n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize();
}
```

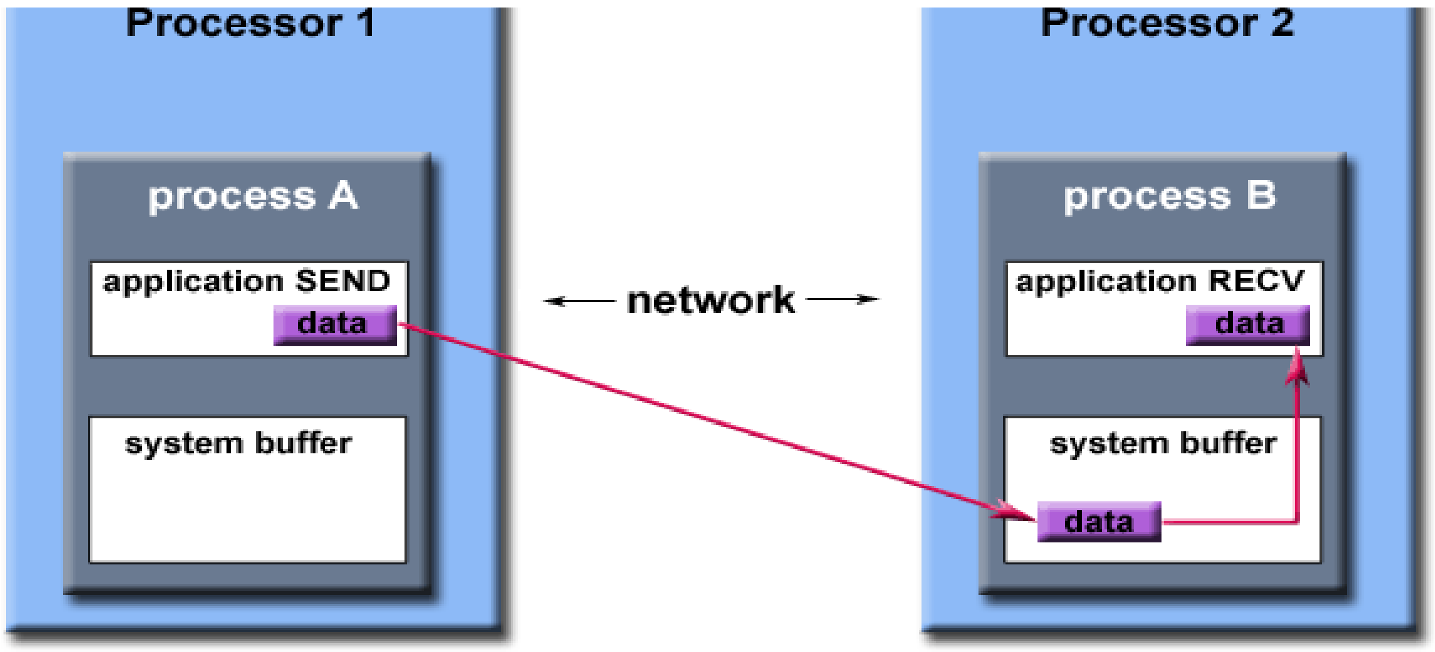
A quick overview of MPI's send modes

Send Modes: represent different choices of **buffering** (where is the data kept until it is received) and **synchronization** (when does a send complete).

- **MPI_Send**
  - MPI_Send will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).
- MPI_Bsend
  - May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- MPI_Ssend
  - will not return until matching receive posted
- MPI_Rsend
  - May be used ONLY if matching receive already posted. User responsible for writing a correct program.
- **MPI_Isend**
  - Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see MPI_Request_free). Note also that while the I refers to immediate, there is no performance requirement on MPI_Isend. An immediate send must return to the user without requiring a matching receive at the destination. An implementation is free to send the data to the destination before returning, as long as the send call does not block waiting for a matching receive. Different strategies of when to send the data offer different performance advantages and disadvantages that will depend on the application.
- MPI_Ibsend
  - buffered nonblocking
- MPI_Issend
  - Synchronous nonblocking. Note that a Wait/Test will complete only when the matching receive is posted.
- MPI_Irsend
  - As with MPI_Rsend, but nonblocking.

Blocking send	MPI_Send(buffer,count,type,dest,tag,comm)
Blocking receive	MPI_Recv(buffer,count,type,source,tag,comm, status)
Blocking Probe	MPI_Probe (source,tag,comm,&status) *-allow checking of incoming messages, without actual receipt of them.* *-If a send that matches the probe has been initiated by some process, then the call to MPI_Probe will return*
Non-blocking send	MPI_Isend(buffer,count,type,dest,tag,comm, request)
Non-blocking receive	MPI_Irecv(buffer,count,type,source,tag,comm, request)
Wait	MPI_Wait (&request,&status)
Test	MPI_Test (&request,&flag,&status)
Non-blocking probe	MPI_Iprobe (source,tag,comm,&flag,&status) *-similar to MPI_Probe but without blocking*

Buffers usage
(decision of the specific MPI implementation)



Path of a message buffered at the receiving process

Additional send functions.
The communication mode is indicated by a one letter prefix: B for buffered, S for synchronous, and R for ready.

Send in buffered mode.

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

IN buf	initial address of send buffer (choice)
IN count	number of elements in send buffer (integer)
IN datatype	datatype of each send buffer element (handle)
IN dest	rank of destination (integer)
IN tag	message tag (integer)
IN comm	communicator (handle)

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

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Send in synchronous mode.

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

IN buf	initial address of send buffer (choice)
IN count	number of elements in send buffer (integer)
IN datatype	datatype of each send buffer element (handle)
IN dest	rank of destination (integer)
IN tag	message tag (integer)
IN comm	communicator (handle)

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

```
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

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

IN buf	initial address of send buffer (choice)
IN count	number of elements in send buffer (integer)
IN datatype	datatype of each send buffer element (handle)
IN dest	rank of destination (integer)
IN tag	message tag (integer)
IN comm	communicator (handle)

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

```
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Receive operations

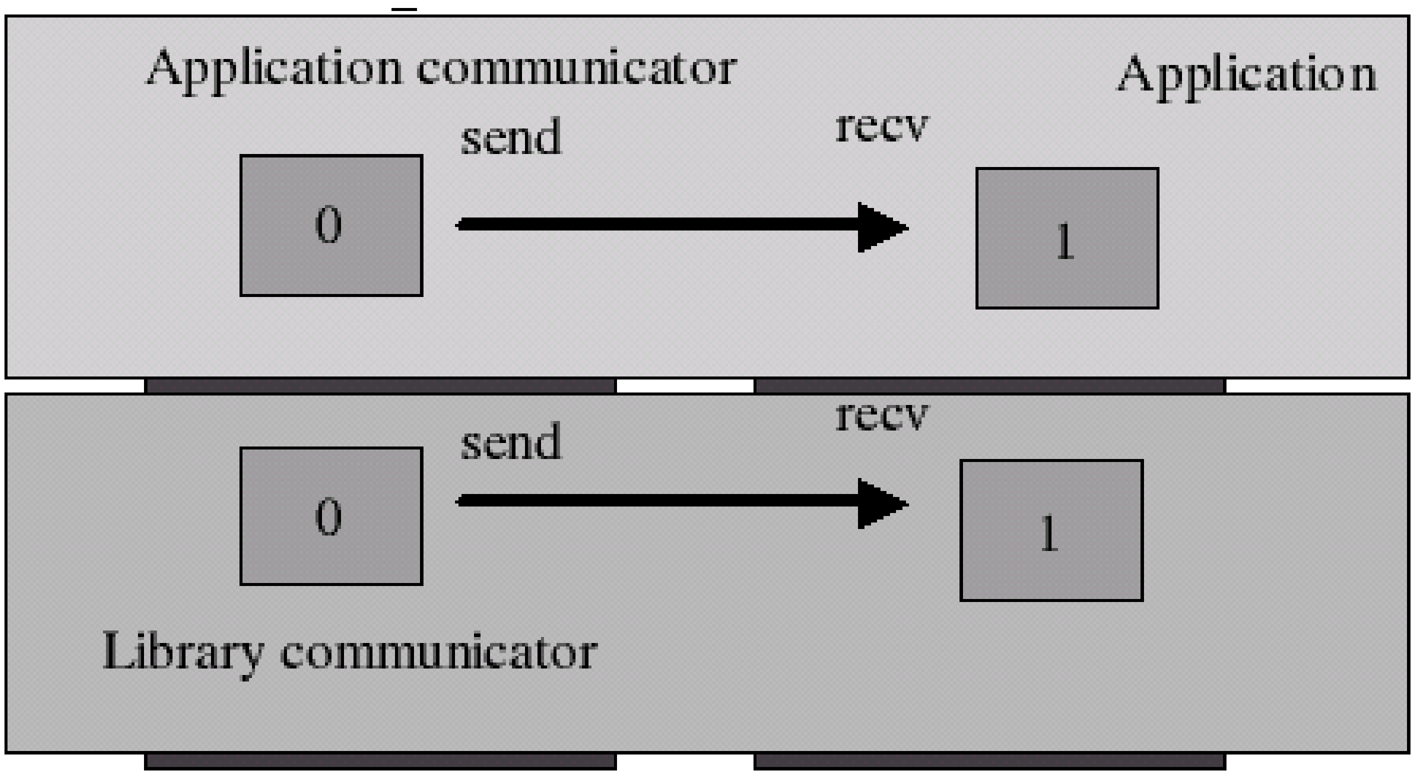
MPI_Recv

- it is blocking op:
    - it returns only after the receive buffer contains the newly received message.

MPI_IRecv

- A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

send-recv


Application communicator
Application
send
recv
0
1
send
recv
0
1
Library communicator

Why Datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very ***different memory representations and lengths of elementary datatypes*** (heterogeneous communication).
- Specifying application-oriented layout of data in memory
  - reduces memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available

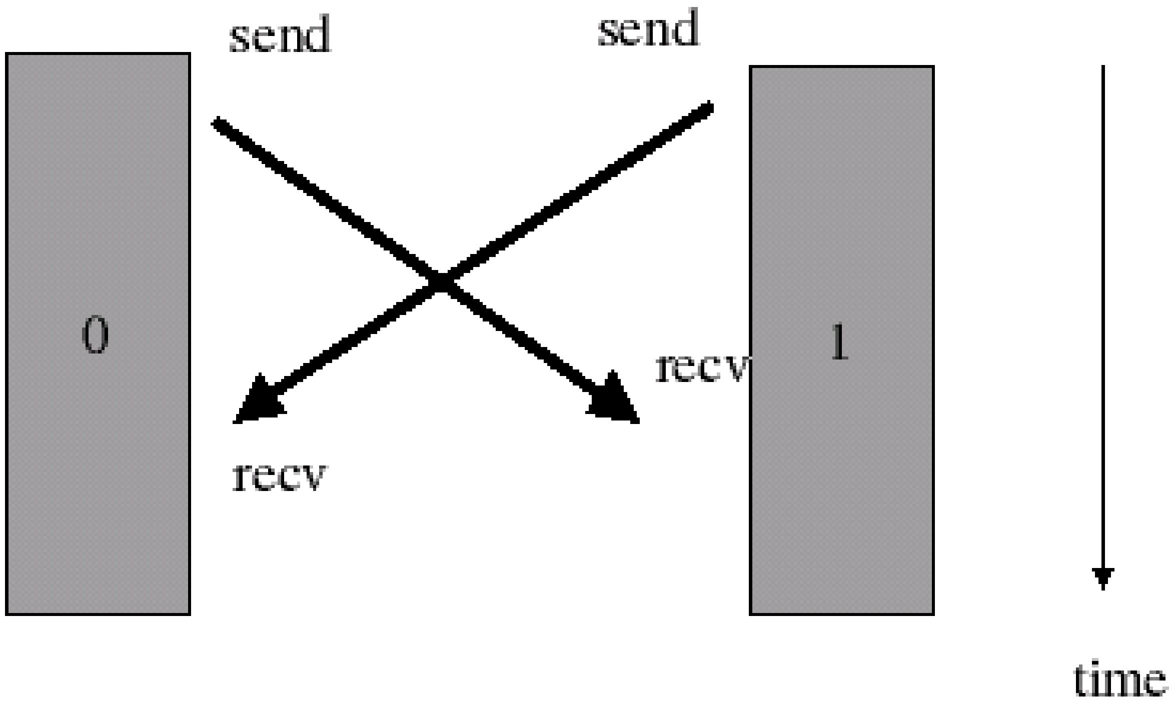
MPI Data Types

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	
MPI_LONG_LONG	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
…	

Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of “wild card” tags.
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application
- Use MPI_Comm_split to create new communicators

Deadlock



Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called "unsafe" because it depends on the availability of system buffers

Deadlock

- Blocking communication may lead to deadlock!
- Incorrect code:

```
 if (rank == 0) {
   dest = source = 1;
   rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
else if (rank == 1){
         dest = source = 0;
         rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
         rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
                                                                &Stat);
         }
```

Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

MPI_Sendrecv

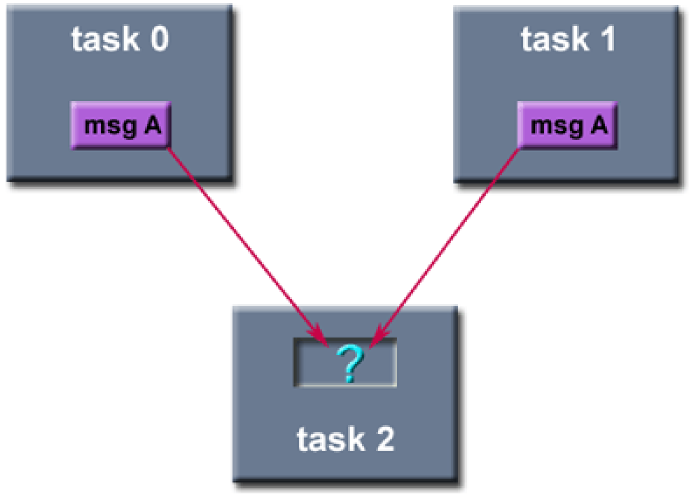
- Send and recv

- *Send a message and post a receive before blocking.*
- *Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.*

MPI_Sendrecv (&sendbuf,sendcount,sendtype, dest, sendtag, &recvbuf, recvcount, recvtype, source, recvtag, comm, &status)

Fairness

- *MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".*
- *Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.*



MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**
- Point-to-point (send/recv) isn’t the only way...

Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

Broadcast – implemented using MPI_Send and MPI_Recv

```
if (id==0){
      for (int i=1;i<nprocs; i++)
          MPI_Send(buf, 1, MPI_CHAR, i, 100, MPI_COMM_WORLD);
}
else // id!=  0
   MPI_Recv(buf, 1, MPI_CHAR, 0, 100, MPI_COMM_WORLD, &status);
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

MPI_Bcast(buf, 1, MPI_CHAR, 0, 100, , MPI_COMM_WORLD);
```

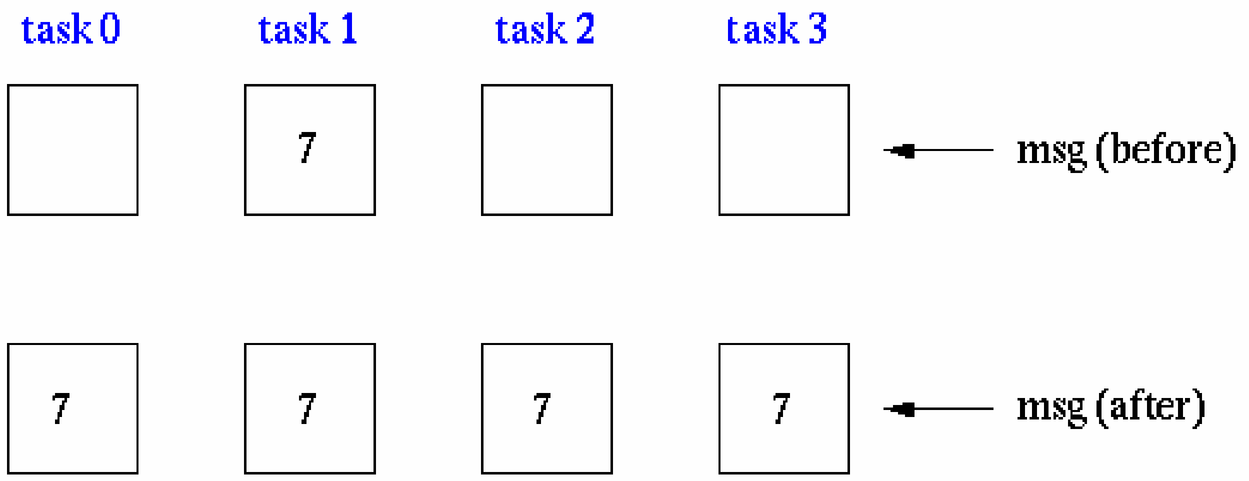
MPI_Barrier

```
MPI_Barrier (comm)
MPI_BARRIER (comm,ierr)
```

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);
```

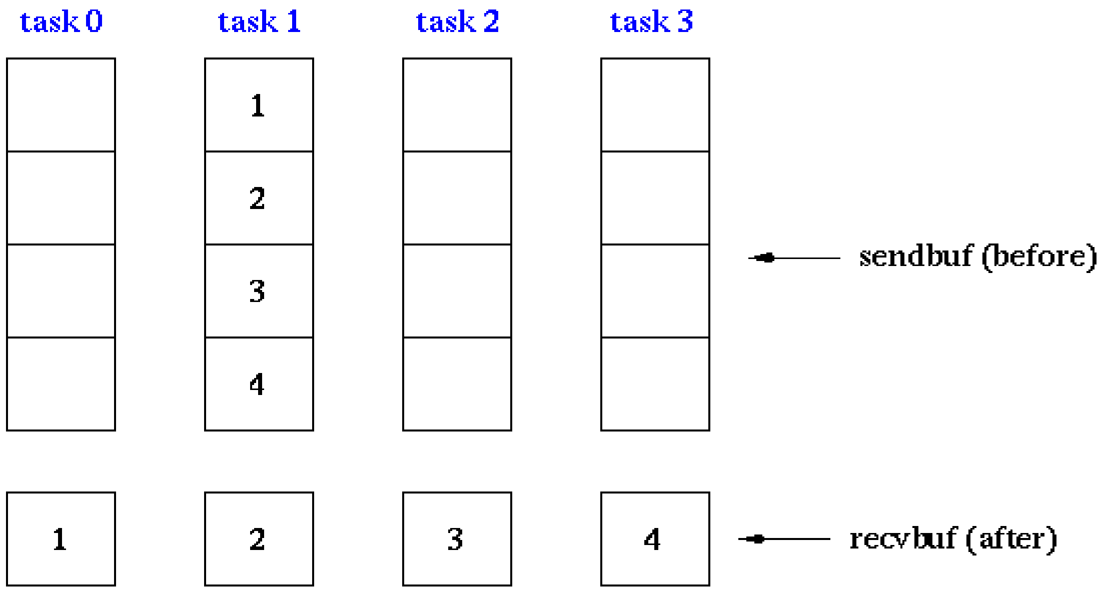


MPI_Scatter

Sends data from one task to all other tasks in a group

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

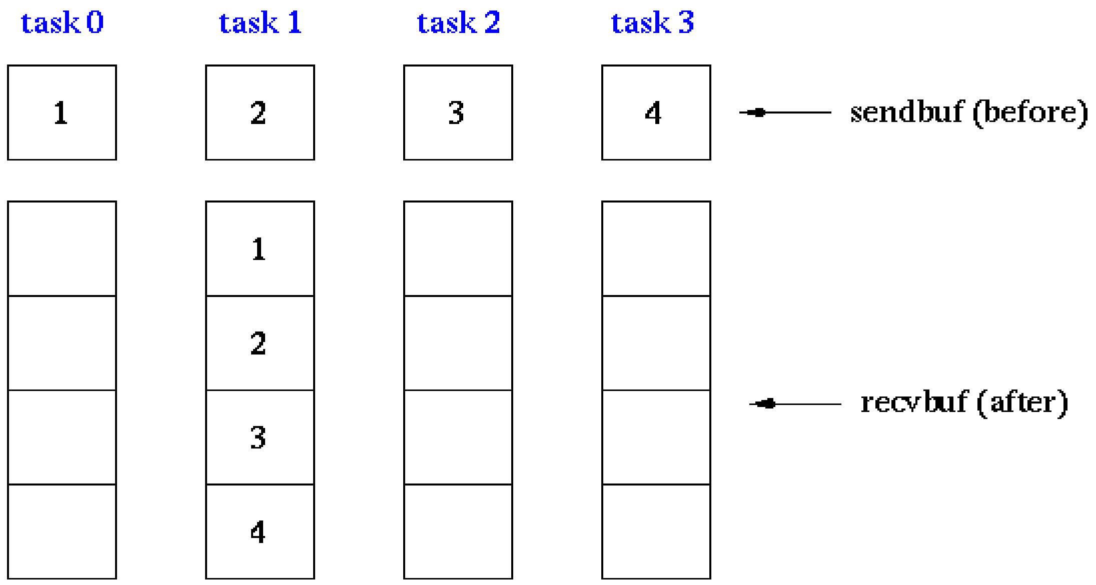
task 1 contains the message to be scattered



MPI_Gather

Gathers together values from a group of processes

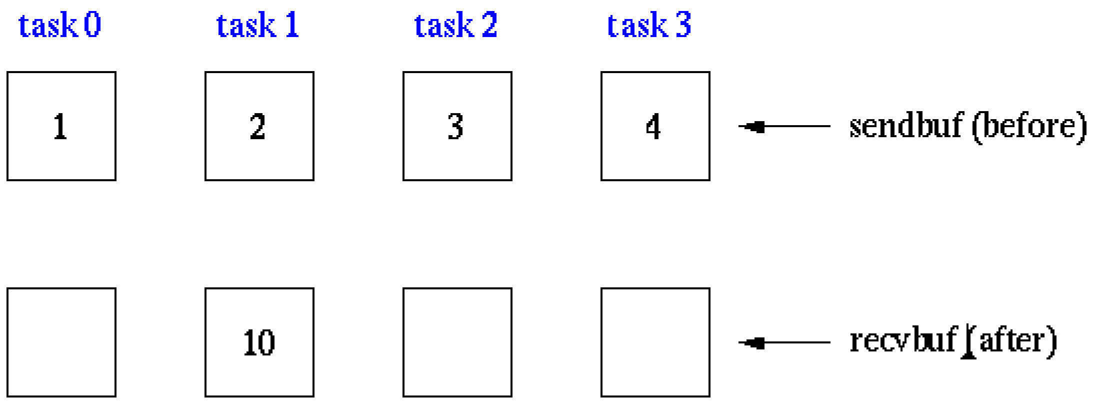
```
sendcnt = 1;
recvcnt = 1;
src  = 1;                 messages will be gathered in task 1
MPI_Gather(sendbuf, sendcnt, MPI_INT,
           recvbuf, recvcnt, MPI_INT,
           src, MPI_COMM_WORLD);
```



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);
```



Collectives' synchronization

- In most libraries, collectives imply a synchronization
  - An implementation without synchronization is costly

- A user program that assumes no synchronization is erroneous
- Incorrect code (High risk of deadlock)

```
if(my_rank == 1)
        MPI_Recv(0);
MPI_Bcast(...);

if(my_rank == 0)
        MPI_Send(1);
```

Example: PI approximation

- The method evaluates the integral of 4/(1+x*x) between 0 and 1.
- The method is simple: the integral is approximated by a sum of n intervals;
  - the approximation to the integral in each interval is (1/n)*4/(1+x*x).
- The master process (rank 0) asks the user for the number of intervals;
  - the master should then broadcast this number to all of the other processes.
- Each process then adds up every n'th interval (x = rank/n, rank/n+size/n,...).
- Finally, the sums computed by each process are added together using a reduction.

Example: PI in C (1)

```
#include "mpi.h"
#include <math.h>
int main(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;
```

Example: PI in C (2)

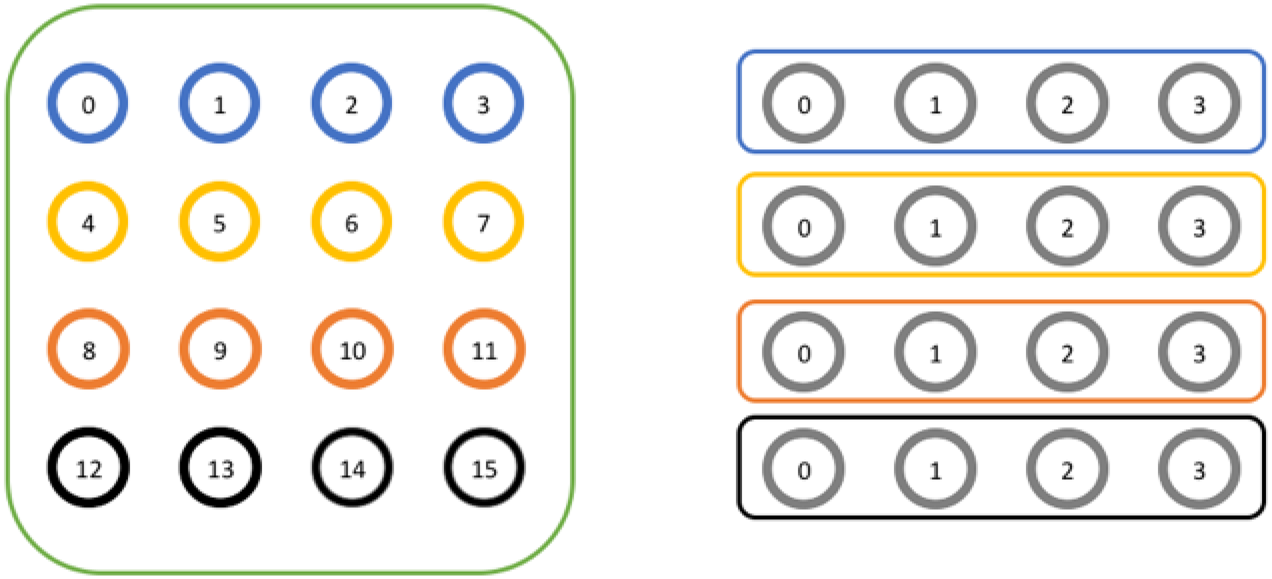
```
    h   = 1.0 / (double) n;
      sum = 0.0;
      for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
      }
      mypi = h * sum;
      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                 MPI_COMM_WORLD);
      if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
                pi, fabs(pi - PI25DT));
    }
    MPI_Finalize();
  return 0;
}
```

The set of 6 Functions for Simplified MPI

  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**
- What else is needed (and why)?

Comunicators

Split a Large Communicator Into Smaller Communicators



Creating new communicators

- Create a new communicators to perform calculations on a subset of the processes in a grid.

```
MPI_Comm_split(
        MPI_Comm comm,
        int color,
        int key,
        MPI_Comm* newcomm)
```

- The color decides to which communicator the process will belong after the split.

Example

```
// Get the rank and size in the original communicator

int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row of 4

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t
        ROW RANK/SIZE: %d/%d\n",
               world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

16 processes

```
WORLD RANK/SIZE: 0/16     ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 1/16     ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 2/16     ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 3/16     ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 4/16     ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 5/16     ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 6/16     ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 7/16     ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 8/16     ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 9/16     ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 10/16    ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 11/16    ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 12/16    ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 13/16    ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 14/16    ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 15/16    ROW RANK/SIZE: 3/4
```

Extensions the Message-Passing Interface

- Dynamic Process Management
  - Dynamic process startup
  - Dynamic establishment of connections
- One-sided communication
  - Put/get
  - Other operations
- Parallel I/O
- Other MPI-2 features
  - Generalized requests
  - Bindings for C++/ Fortran-90; interlanguage issues

Some Simple Exercises

- Compile and run the **`hello`** and **`pi`** programs.
- Modify the **`pi`** program to use send/receive instead of bcast/reduce.
- Write a program that sends a message around a ring. That is, process 0 reads a line from the terminal and sends it to process 1, who sends it to process 2, etc. The last process sends it back to process 0, who prints it.
- Time programs with **`MPI_WTIME`**. (Find it.)

When to use MPI

- Appropriate when we need:
  - Portability and Performance
  - Building Tools for Others
    - Libraries
  - Need to Manage memory on a per processor basis

- When *not* appropriate to use MPI
  - Require Fault Tolerance
    - Sockets
  - Distributed Computing
    - CORBA, DCOM, etc.

Implementations

- **MPICH –**
- **Open MPI –**
- **IBM MPI –**

- **IntelMPI (not free)**

Summary

- The parallel computing community has cooperated on the development of a standard for message-passing libraries.
- There are many implementations, on nearly all platforms.
- MPI subsets are easy to learn and use.
- Lots of MPI materials are available.