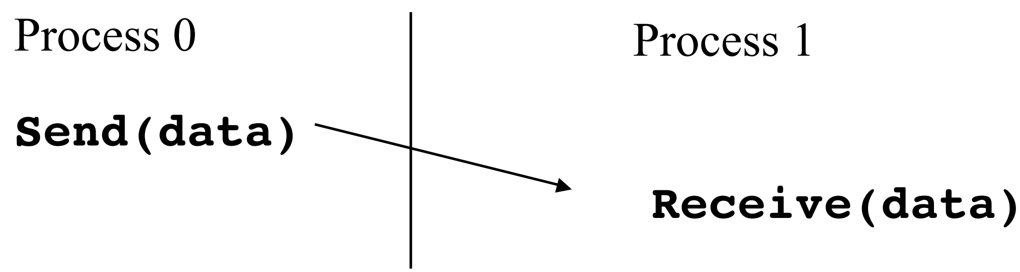
MPI Basic Send/Receive

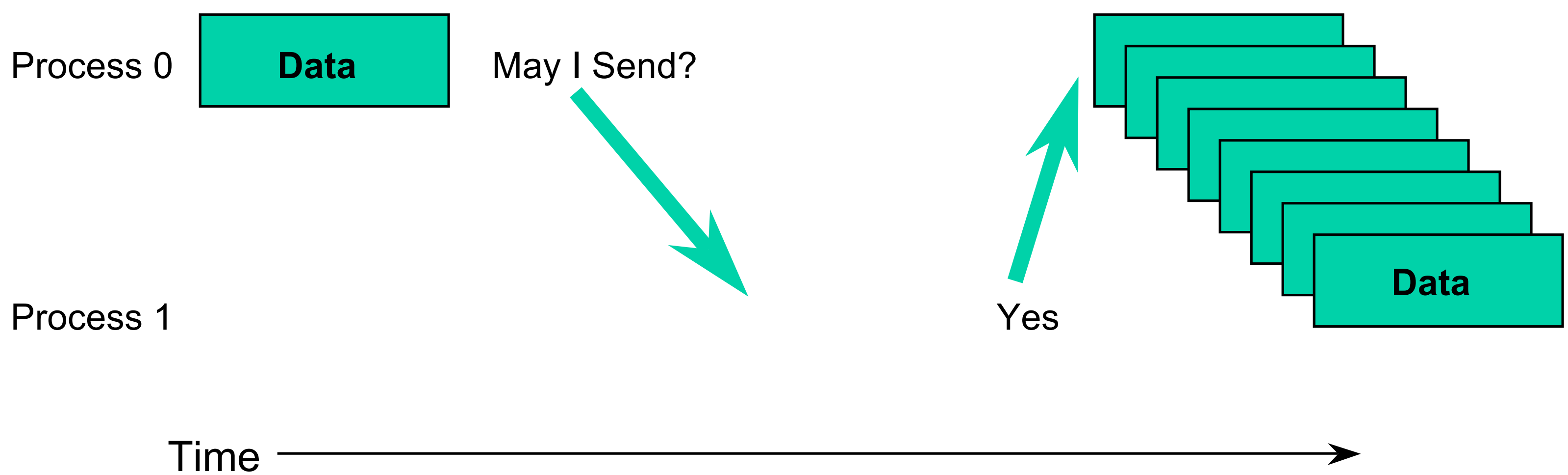
- We need to fill in the details in



- 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

MPI Data Types

- 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; //actual tag
recvd_from = status.MPI_SOURCE; //actual source
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Why Data Types?

- 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 from PowerPC processor to Intel's).
- 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

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

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(dst=1)`	`Send(dst=0)`
`Recv(src=1)`	`Recv(src=0)`

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

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`**

Nonblocking Send/Receive

- How to send a large message without knowing the limitation of a computer system?
  - Use nonblocking send, MPI_Isend, and follow later with MPI_Wait
    - This can send an arbitrarily large message
    - Improve performance by not making an internal copy

Nonblocking Send/Receive (continue)

- MPI_SEND/MPI_RECV are *blocking* operations
  - A"Send" operation do not complete until matching "Receive" is issued on the destination process
- Moving and manipulating data within a single process is much faster than moving data from one process to another
  - MPI solution: nonblocking send and receive
    - Permit a program to continue to execute or compute instead of waiting for communications to complete

1GHz CPU
(1 *billion* floating point operations per second)

A hundred million bytes per second

1GHz CPU
(1 *billion* floating point operations per second)

Nonblocking Send/Receive (continue)

- Library for nonblocking send/receive
  - MPI_ISEND(buffer, count, datatype, dest, tag, comm, request, ierr)
  - MPI_IRECV(buffer, count, datatype, source, tag, comm, request, ierr)
  - MPI_WAIT(request, status, ierr)
  - MPI_WAITALL(count, array_of_requests, array_of_status, ierr)
- Example:

  call MPI_IRECV(…, requests(1), ierr)

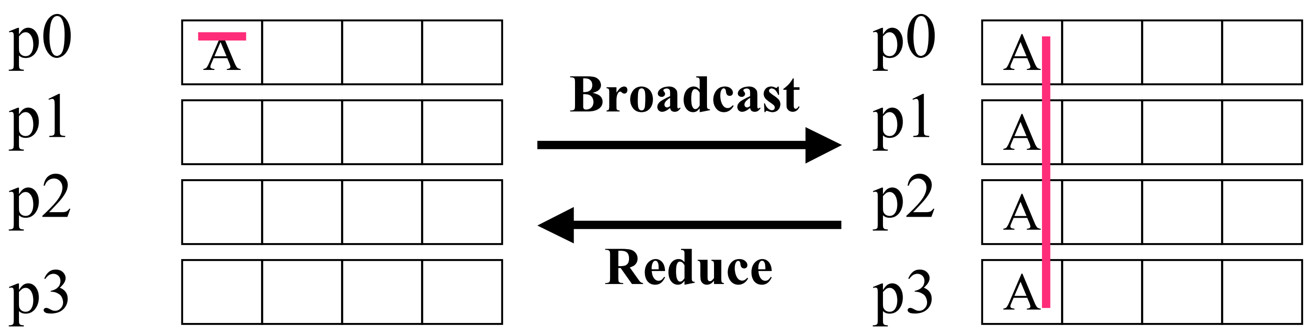
  call MPI_IRECV(…, requests(2), ierr)

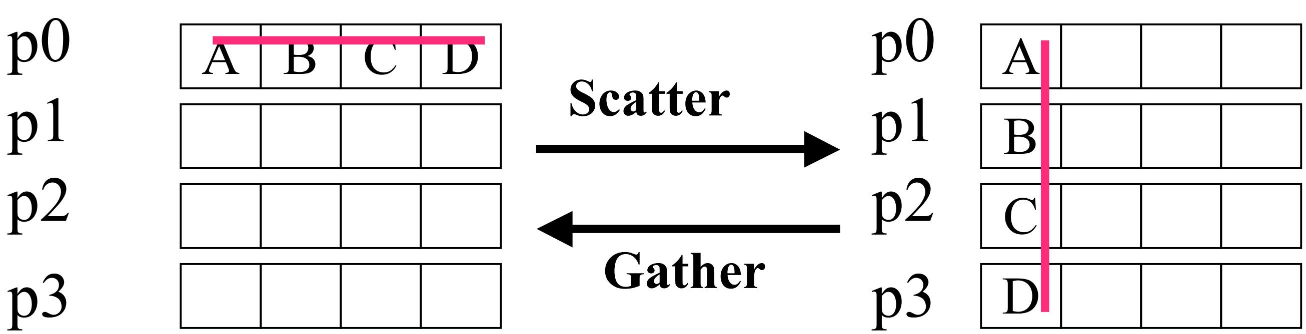
  …

  call MPI_WAITALL(2, requests, status, ierr)

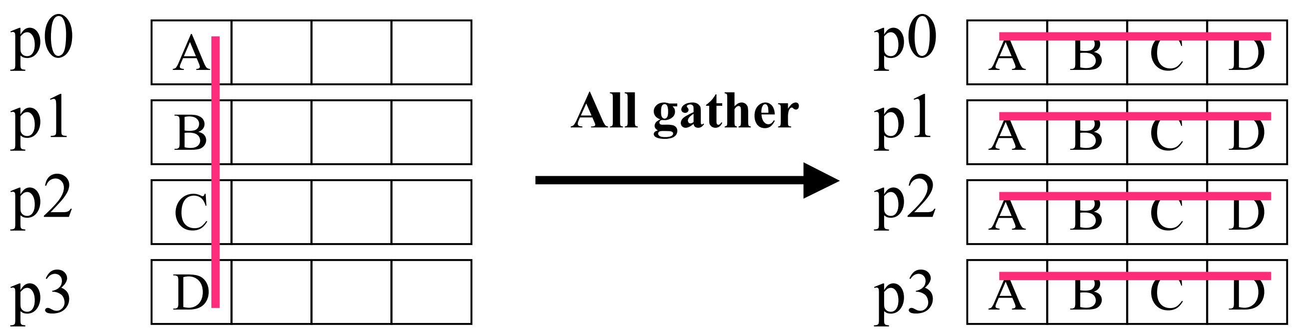
  where "status" is declared as "integer status (MPI_STATUS_SIZE, 2)"
- More in the example of solving Poisson problems

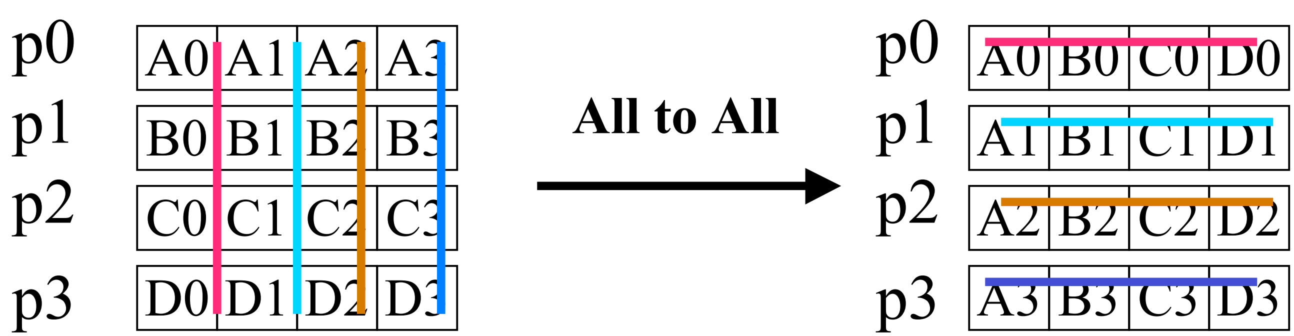
Collective Operations





Advanced Collective Operations (continue)





Distributed Computing

- MPI is not suitable for distributed computing
  - The intent of MPI is to support tightly-coupled parallel computing, where all processes are working on the same or a similar problem
  - MPI-1 contains no support for the client-server model available in CORBA and DCOM
  - MPI-2 provide limited support for client-server model
    - OpenMPI implementation appears to support this feature
- Grid-enabled MPI is intended to perform message-passing operations via Grid
  - Appear to be inactive now

Monte Carlo Method

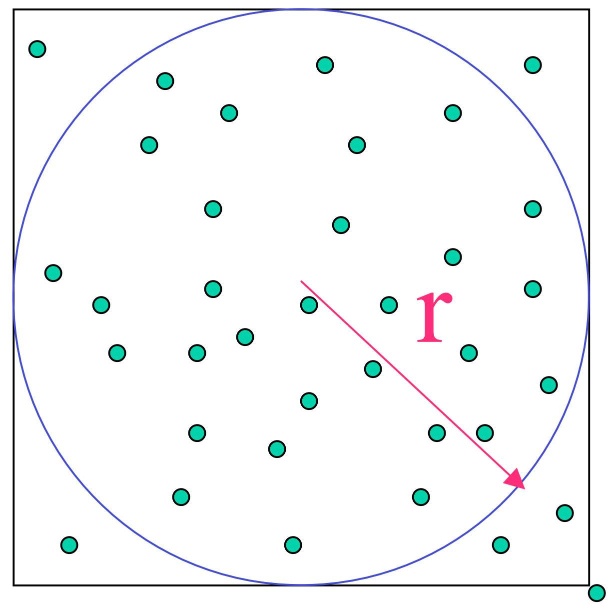
- A class of computational algorithms that rely on repeated random sampling to compute their results
- The term was coined in the 1940s by physicists working on nuclear weapon projects in the Los Alamos National Laboratory
- Tend to be used when it is infeasible or impossible to compute an exact result with a deterministic algorithm such as molecular dynamics.
- Especially useful in studying systems with a large number of coupled degrees of freedom, such as fluids, disordered materials, strongly coupled solids, and cellular structures.
- More broadly, useful for modeling phenomena with significant uncertainty in inputs, such as the calculation of risk in business.
- In finance and mathematical finance, used to value and analyze (complex) instruments, portfolios and investments by simulating the various sources of uncertainty affecting their value, and then determining their average value over the range of resultant outcomes.
  - Its advantage increases as the dimensions (sources of uncertainty) of the problem increase.

Monte-Carlo Method (Algorithm)

1) Specify an initial points x0 in phase space.
2) Generate a new state x'.
3) Compute the transition probability W(x,x')
4) Generate a uniform random number R between [0,1]
5) If w<R, then return to step 2
6) Otherwise, accept the new state and return to step 2

Monte Carlo computation of pi

- Square area = 2r * 2r
- Circle area = pi r*r
- Pi = 4*circle area/square area
- Selection criterion: A particle inside the circle or not
- The area value is represented with number of dots
- Radius = 1 for convenience
- In computation, “dimensionless variables are typically used



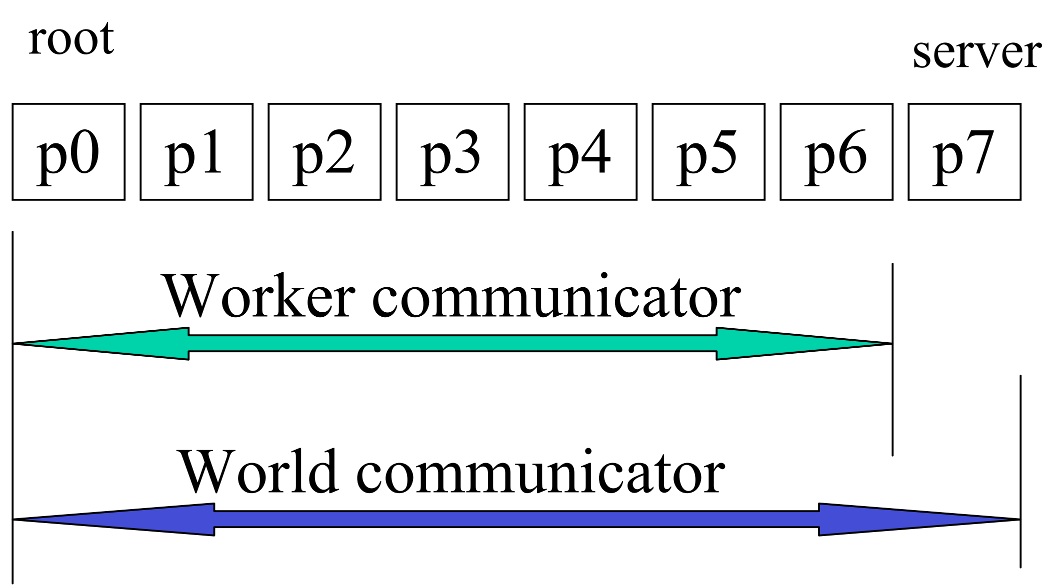
Monte Carlo Computation of pi (1)

```
/* compute pi using Monte Carlo method */
#include <math.h>
#include "mpi.h"
#include "mpe.h"
#define CHUNKSIZE      1000
/* We'd like a value that gives the maximum value returned by the function
   random, but no such value is *portable*.  RAND_MAX is available on many
   systems but is not always the correct value for random (it isn't for
   Solaris).  The value ((unsigned(1)<<31)-1) is common but not guaranteed */

#define INT_MAX 1000000000
/* message tags */
#define REQUEST  1
#define REPLY    2
```

Monte Carlo Computation of pi (2)

```
int main( int argc, char *argv[] )
{
   int iter;
   int in, out, i, iters, max, ix, iy, ranks[1], done, temp;
   double x, y, Pi, error, epsilon;
   int numprocs, myid, server, totalin, totalout, workerid;
   int rands[CHUNKSIZE], request;
   MPI_Comm world, workers;
   MPI_Group world_group, worker_group;
   MPI_Status status;

   MPI_Init(&argc,&argv);
   world  = MPI_COMM_WORLD;
   MPI_Comm_size(world, &numprocs);
   MPI_Comm_rank(world, &myid);
   server = numprocs-1;              /* last proc is server */
```

Monte Carlo Computation of pi (0)



Monte Carlo Computation of pi (3)

```
if (myid == 0)
     sscanf( argv[1], "%lf", &epsilon );   /* input precision */

  MPI_Bcast( &epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD );
  MPI_Comm_group( world, &world_group ); /* create a group of world communicator*/
  ranks[0] = server;

/* Create a worker group  by excluding the server process from world group*/
  MPI_Group_excl( world_group, 1, ranks, &worker_group );  //ranks is the address for server

 /* Create a new communicator, workers*/
  MPI_Comm_create( world, worker_group, &workers );

  MPI_Group_free(&worker_group);
```

Monte Carlo Computation of pi (4)

```
if (myid == server) { /* I am the server process for generating random number*/
        do {
          MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST,
                   world, &status);
          if (request) {                              /* request is valid */
                for (i = 0; i < CHUNKSIZE; )
                {
                    rands[i] = random();
                    if (rands[i] <= INT_MAX) i++;
                }
                MPI_Send(rands, CHUNKSIZE, MPI_INT,
            status.MPI_SOURCE, REPLY, world);
          }
        }
        while( request>0 );
 }
```

Monte Carlo Computation of pi (5)

```
else {                              /* I am one of the worker processes */
     request = 1;
             done = in = out = 0;
             max  = INT_MAX;        /* max int, for normalization */
     MPI_Send( &request, 1, MPI_INT, server, REQUEST, world );
     MPI_Comm_rank( workers, &workerid ); /* get a worker's rank */
             iter = 0;
             while (!done) {
                iter++;
                request = 1;
                MPI_Recv( rands, CHUNKSIZE, MPI_INT, server, REPLY,
                               world, &status );
               //process the random numbers of CHUNKSIZE
               for (i=0; i<CHUNKSIZE-1; ) {
                   x = (((double) rands[i++])/max) * 2 - 1;
                   y = (((double) rands[i++])/max) * 2 - 1;
                           if (x*x + y*y < 1.0)
                              in++;
                           else
                              out++;
                }
```

Monte Carlo Computation of pi (6)

```
MPI_Allreduce(&in, &totalin, 1, MPI_INT, MPI_SUM,
                              workers);
MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM,
                              workers);
Pi = (4.0*totalin)/(totalin + totalout);
error = fabs( Pi-3.141592653589793238462643);
done = (error < epsilon || (totalin+totalout) > 1000000);
request = (done) ? 0 : 1;   /*??  */

//continue the operations based on the value of request
if (myid == 0) {
            printf( "\rpi = %23.20f", Pi );
            MPI_Send( &request, 1, MPI_INT, server, REQUEST,
                              world );
}
else {
            if (request)
               MPI_Send(&request, 1, MPI_INT, server, REQUEST,
                              world);
}
```

Monte Carlo Computation of pi (7)

```
        }  /* end of “while (!done)”
        MPI_Comm_free(&workers);
  }  /* end of  “myid == server”

  if (myid == 0) {
     printf( "\npoints: %d\nin: %d, out: %d, <ret> to exit\n",
              totalin+totalout, totalin, totalout );

        getchar();  /* clear garbage character */
  }
  MPI_Finalize();
}
```