Parallel Programming Principle and Practice

Lecture 12-2 —parallel programming :

Message Passing Paradigm MPI

華中科技大學

Outline

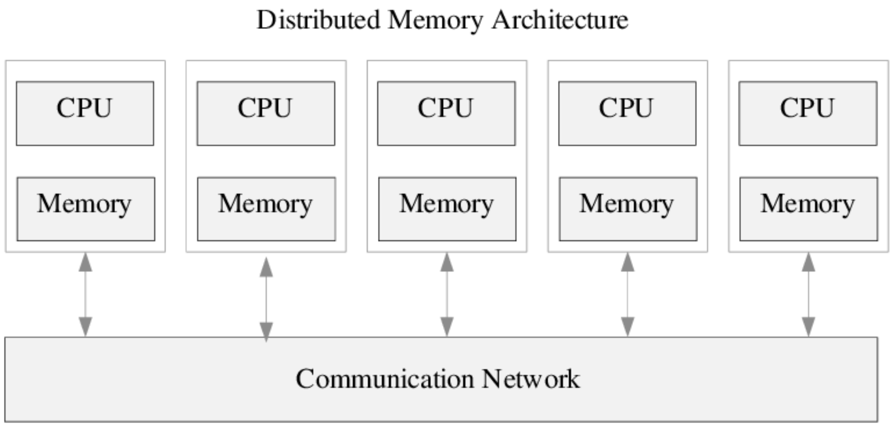
- Message Passing Paradigm MPI
  - Message-Passing Programming Model
  - An overview of MPI programming
    - Six MPI functions and hello sample
    - How to compile/run
  - Send/Receive communication
    - Application Example: Parallelizing numerical integration with MPI
  - Collective group communication
    - Application Examples: Pi computation
  - Safety Issues in MPI programs

MPI

Message-Passing Programming Model

Recall Programming Model： Distributed Memory(Message-Passing)

- **distributed memory** refers to a multiprocessor computer system in which each processor has its own private memory
- a **shared memory** multiprocessor offers a single memory space used by all processors
  - Computational tasks can only operate on local data, and if remote data are required, the computational task must communicate with one or more remote processors
  - **MPI** is designed **Mainly** for distributed memory systems



MPI

An overview of MPI programming

What is MPI ?

- **Message Passing Interface (MPI)** is a standardized and portable message-passing standard designed to function on parallel computing architectures
- The MPI standard defines the syntax and semantics of **library routines** that are useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran
- There are **several open-source MPI implementations**, which fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications
- The **Standard** itself (**MPI-2**, **MPI-3**):
  - at http://www.mpi-forum.org

Six MPI functions

- 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

Which two?

- • To measure time: MPI_Wtime()

Simple Example: Hello World!

- **A classic type of C**

```
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```

Simple Example: Hello World!

- **Mpi_hello (C)**

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

int main( 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 );
    printf( "I am %d of %d!\n", rank, size )
    MPI_Finalize();
    return 0;
}
```

How to compile and run ?

- How to compile?

wrapper script to compile

source file

mpicc -g -Wall -o mpi_hello mpi_hello.c

produce debugging information

create this executable file name (as opposed to default a.out)

turns on all warnings

How to compile and run ?

- How to run : Execution with mpirun

mpirun -n <number of processes> <executable>

mpirun -n 1 ./mpi_hello

run with 1 process

mpirun -n 4 ./mpi_hello

run with 4 processes

Execution

```
mpirun -n 1 ./mpi_hello

    I am 0 of 1 !
```

```
mpirun -n 4 ./mpi_hello

    I am 0 of 4 !
    I am 1 of 4 !
    I am 2 of 4 !
    I am 3 of 4 !
```

MPI Programs

- Written in C/C++
  - Has main
  - Uses stdio.h, string.h, etc.
- Need to add **mpi.h** header file
- Identifiers defined by MPI start with "**MPI_**"
- First letter following underscore is **uppercase**
  - For function names and MPI-defined types
  - Helps to avoid confusion
- MPI functions return error codes or **MPI_SUCCESS**

MPI Components

□ MPI_Init

- ➢ Tells MPI to do all the necessary setup

```
int MPI_Init(
        int*      argc_p  /* in/out */,
        char***   argv_p  /* in/out */);
```

□ MPI_Finalize

- ➢ Tells MPI we're done, so clean up anything allocated for this program

```
int MPI_Finalize(void);
```

Basic Outline

- The basic framework of MPI code

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
    . . .
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);
    . . .
    MPI_Finalize();
    /* No MPI calls after this */
    . . .
    return 0;
}
```

MPI

Send/Receive communication

Basic Concepts: Communicator

- Processes can be collected into groups
  - **Communicator**
  - Each message is sent & received in the same 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

Communicators

```
int MPI_Comm_size(
      MPI_Comm  comm       /* in  */,
      int*      comm_sz_p  /* out */);
```

number of processes in the communicator

```
int MPI_Comm_rank(
      MPI_Comm  comm       /* in  */,
      int*      my_rank_p  /* out */);
```

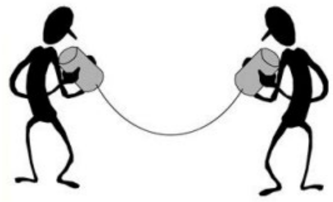
my rank
(the process making this call)

Basic Send

```
int MPI_Send(
   void*          msg_buf_p      /* in */,
   int            msg_size       /* in */,
   MPI_Datatype   msg_type       /* in */,
   int            dest           /* in */,
   int            tag            /* in */,
   MPI_Comm       communicator   /* in */);
```

- Where
  - msg_buf_p is the address of the message
  - msg_size is the quantity of content
  - msg_type is the data type of the message content
  - dest is the target process number
  - tag is the message flag
  - communicator is the communication domain.

- Things that need specifying
  - How will "data" be described? (msg_buf , msg_size, msg_type)
  - How will target processes be identified? (dest)
  - How will the receiver recognize messages? (tag, communicator)



Data types

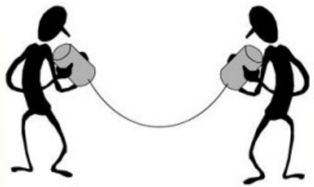
MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned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

Basic Receive: Block until a matching message is received

```
int MPI_Recv(
        void*           msg_buf_p     /* out  */,
        int             buf_size      /* in   */,
        MPI_Datatype    buf_type      /* in   */,
        int             source        /* in   */,
        int             tag           /* in   */,
        MPI_Comm        communicator  /* in   */,
        MPI_Status*     status_p      /* out  */);
```

- Where :
  - msg_buf_p is the address where messages are stored
  - buf_size is number of received messages
  - buf_type is the data type of the message content
  - source is the process number of the receiving source
  - tag is the message flag
  - communicator is the communication domain
  - status is the receiving status

- Things that need specifying:
  - Where to receive data?(msg_buf_p)
  - How will the receiver recognize/screen messages?(source, tag, communicator)



華中科技大學

Message matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

MPI_Send

dest

MPI_Recv

src

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

Message matching

- A receiver can get a message without knowing
  - the sender of the message
    - – Specify the source as MPI_ANY_SOURCE
  - or the tag of the message
    - – Specify the tag as MPI_ANY_TAG
- Status argument: who sent me and what tag is?

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

MPI_Status*

Who sent me
What tag is

Retrieving Further Information from status argument in C/C++

- 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 );
```

- In C++:

```
int recvd_tag, recvd_from, recvd_count;
MPI::Status status;
Comm.Recv(..., MPI::ANY_SOURCE, MPI::ANY_TAG, ...,
          status )
recvd_tag   = status.Get_tag();
recvd_from  = status.Get_source();
recvd_count = status.Get_count( datatype );
```

MPI_Wtime()

- Returns the current time with a double float
- To time a program segment
  - Start time= MPI_Wtime()
  - End time = MPI_Wtime()
  - Time spent is end_time – start_time
- Example of using MPI_Wtime()

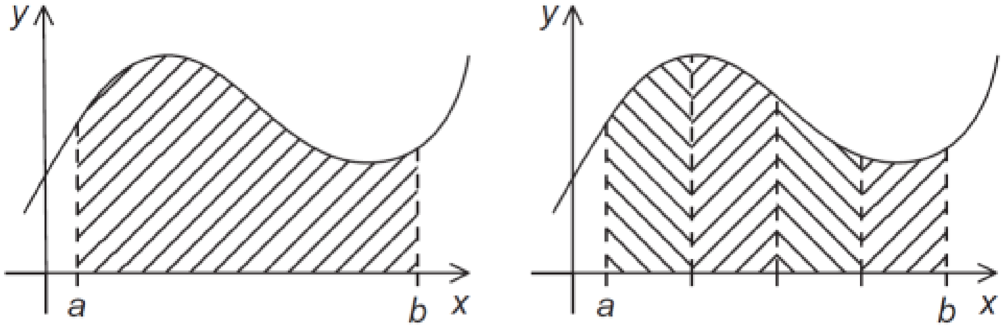
```
#include<stdio.h>
#include<mpi.h>
main(int argc, char **argv){
  int size, node;            double start, end;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &node);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  start = MPI_Wtime();
  if(node==0)  {
      printf(" Hello From Master. Time = %lf \n", MPI_Wtime() -
   start);
}
  else  {
      printf("Hello From Slave #%d %lf \n", node, (MPI_Wtime()
   - start));
    }
  MPI_Finalize();
}
```

MPI

Application Example: Parallelizing numerical integration with MPI

Numerical Integration

- How to compute the Numerical Integration?



(a) (b)

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \cdots\cdots + \int_{x_{n-1}}^{x_n} f(x)dx$$

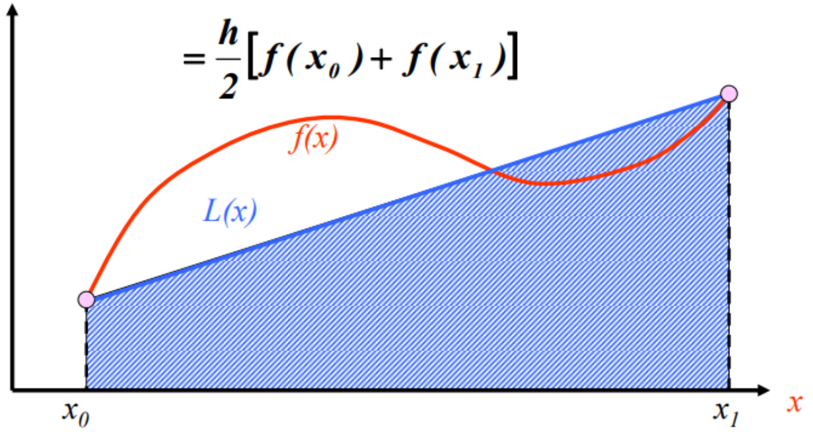
Numerical Integration: idea-1

- Use a simple function to approximate the integral area : **Trapezoid Rule**

- Straight-line approximation

$$\int_a^b f(x)dx \approx \sum_{i=0}^{1} c_i f(x_i) = c_0 f(x_0) + c_1 f(x_1)$$

$$= \frac{h}{2}[f(x_0) + f(x_1)]$$



- Evaluate the integral

Evaluate the integral $\int_0^4 xe^{2x}dx$

- Exact solution

$$\int_0^4 xe^{2x}dx = \left[\frac{x}{2}e^{2x} - \frac{1}{4}e^{2x}\right]_0^4$$

$$= \frac{1}{4}e^{2x}(2x-1)\Big|_0^1 = 5216.926477$$

- Trapezoidal Rule

$$I = \int_0^4 xe^{2x}dx \approx \frac{4-0}{2}[f(0)+f(4)] = 2(0+4e^8) = 23847.66$$

$$\varepsilon = \frac{5216.926 - 23847.66}{5216.926} = -357.12\%$$

Numerical Integration: idea-2

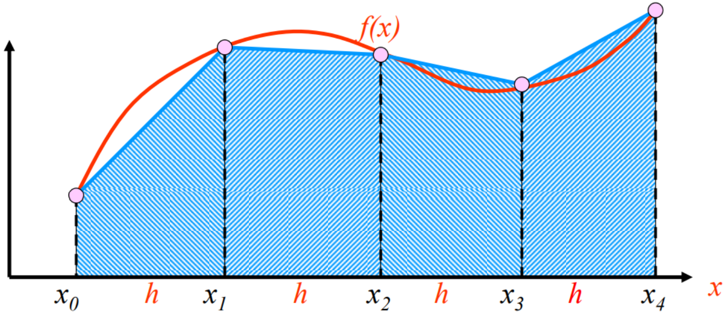
- Apply trapezoid rule to multiple segments : Composite Trapezoid Rule

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \cdots\cdots + \int_{x_{n-1}}^{x_n} f(x)dx$$

$$= \frac{h}{2}[f(x_0) + f(x_1)] + \frac{h}{2}[f(x_1) + f(x_2)] + \cdots + \frac{h}{2}[f(x_{n-1}) + f(x_n)]$$

$$= \frac{h}{2}[f(x_0) + 2f(x_1) + \cdots + 2f(x_i) + \cdots + 2f(x_{n-1}) + f(x_n)]$$

$$h = \frac{b-a}{n}$$



- Evaluate the integral

Evaluate the integral $\boxed{I = \int_0^4 xe^{2x}dx}$

$$n = 1, h = 4 \Rightarrow I = \frac{h}{2}[f(0) + f(4)] = 23847.66 \qquad \varepsilon = -357.12\%$$

$$n = 2, h = 2 \Rightarrow I = \frac{h}{2}[f(0) + 2f(2) + f(4)] = 12142.23 \qquad \varepsilon = -132.75\%$$

$$n = 4, h = 1 \Rightarrow I = \frac{h}{2}[f(0) + 2f(1) + 2f(2) + 2f(3) + f(4)] = 7288.79 \qquad \varepsilon = -39.71\%$$

$$n = 8, h = 0.5 \Rightarrow I = \frac{h}{2}[f(0) + 2f(0.5) + 2f(1) + 2f(1.5) + 2f(2) + 2f(2.5) + 2f(3) + 2f(3.5) + f(4)] = 5764.76 \qquad \varepsilon = -10.50\%$$

$$n = 16, h = 0.25 \Rightarrow I = \frac{h}{2}[f(0) + 2f(0.25) + 2f(0.5) + \cdots + 2f(3.5) + 2f(3.75) + f(4)] = 5355.95 \qquad \varepsilon = -2.66\%$$

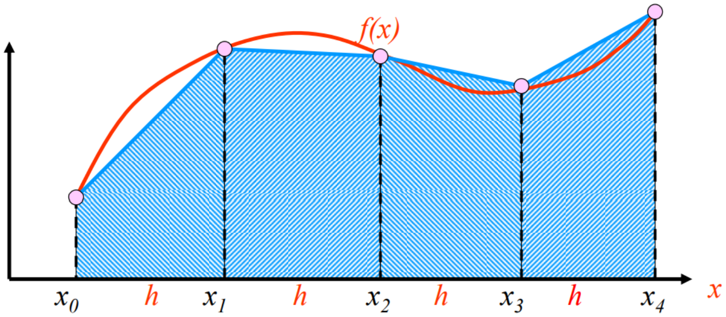
Numerical Integration: Pseudo-code for a serial program

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \cdots\cdots + \int_{x_{n-1}}^{x_n} f(x)dx$$

$$= \frac{h}{2}[f(x_0) + f(x_1)] + \frac{h}{2}[f(x_1) + f(x_2)] + \cdots + \frac{h}{2}[f(x_{n-1}) + f(x_n)]$$

$$= \frac{h}{2}[f(x_0) + 2f(x_1) + \cdots + 2f(x_i) + \cdots + 2f(x_{n-1}) + f(x_n)]$$

$$\boxed{h = \frac{b-a}{n}}$$

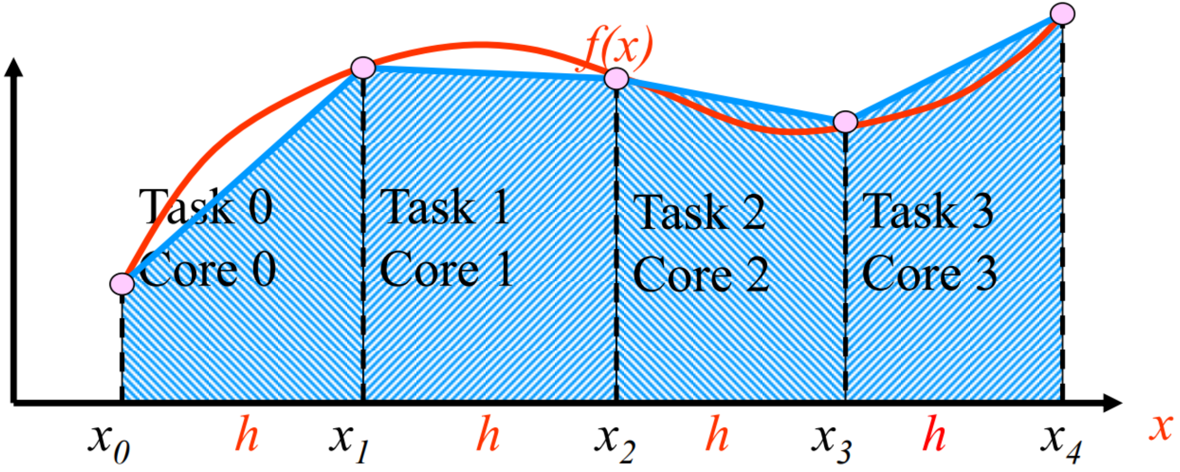



```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 0; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

Numerical Integration: Parallelizing the Trapezoidal Rule

- Partition problem solution into tasks
- Identify communication channels between tasks
- Aggregate tasks into composite tasks
- Map

$$\boldsymbol{h = \frac{b-a}{n}}$$



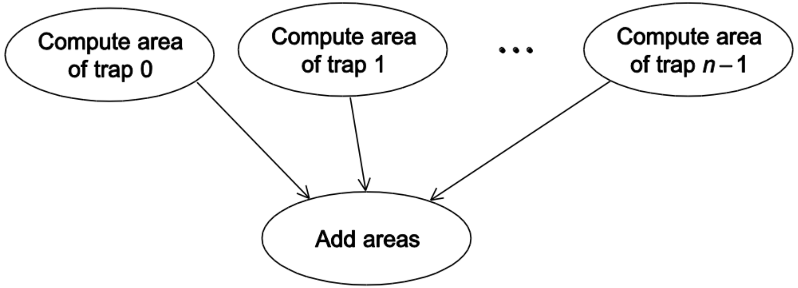
Numerical Integration: Parallel pseudo-code

```
Get a, b, n;
h = (b-a)/n;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0)
   Send local_integral to process 0;
else /* my_rank == 0 */
   total_integral = local_integral;
   for (proc = 1; proc < comm_sz; proc++) {
      Receive local_integral from proc;
      total_integral += local_integral;
   }
}
if (my_rank == 0)
   print result;
```

Compute the local area

Summation of local values

- comm_sz : number of processes in the communicator
- Trap() : Compute the integration between lacal_a and local_b with local_n segments
- <u>process 0 sum all the integration</u>



Numerical Integration: Parallel MPI code

Parallel MPI code version

```
int main(void) {
   int my_rank, comm_sz, n = 1024, local_n;
   double a = 0.0, b = 3.0, h, local_a, local_b;
   double local_int, total_int;
   int source;

   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   h = (b-a)/n;           /* h is the same for all processes */
   local_n = n/comm_sz;   /* So is the number of trapezoids  */

   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   local_int = Trap(local_a, local_b, local_n, h);

   if (my_rank != 0) {
      MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
            MPI_COMM_WORLD);
```

Use send/receive to sum

```
   } else {
      total_int = local_int;
      for (source = 1; source < comm_sz; source++) {
         MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         total_int += local_int;
      }
   }
```

Use send/receive to sum

```
   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }
   MPI_Finalize();
   return 0;
} /* main */
```

Numerical Integration: Parallel MPI code

MPI version

- Trap function is still a serial program

```
double Trap(
      double left_endpt  /* in */,
      double right_endpt /* in */,
      int    trap_count  /* in */,
      double base_len    /* in */) {
   double estimate, x;
   int i;

   estimate = (f(left_endpt) + f(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
      x = left_endpt + i*base_len;
      estimate += f(x);
   }
   estimate = estimate*base_len;

   return estimate;
} /* Trap */
```

MPI

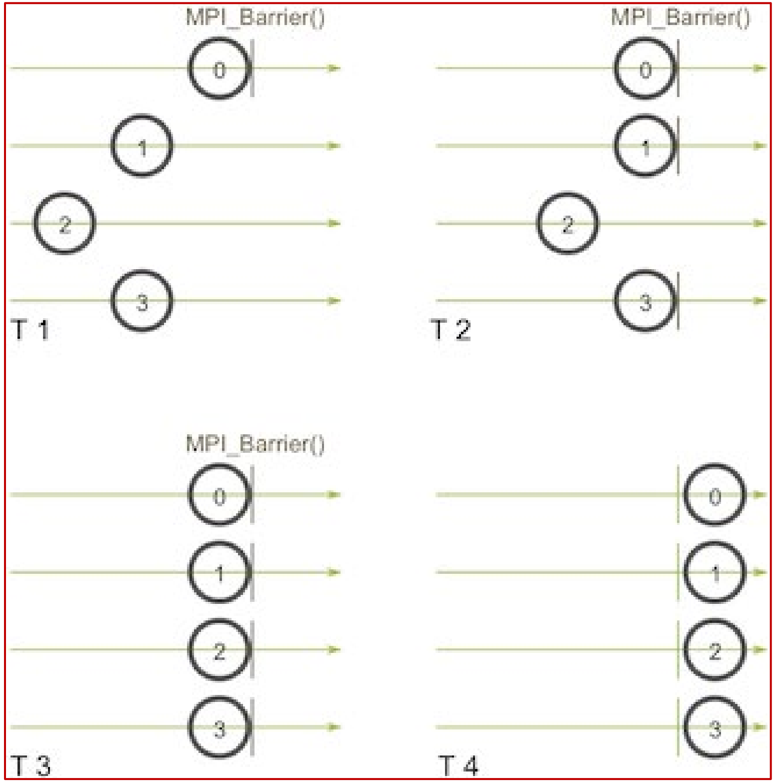
Collective group communication

MPI Collective Communication

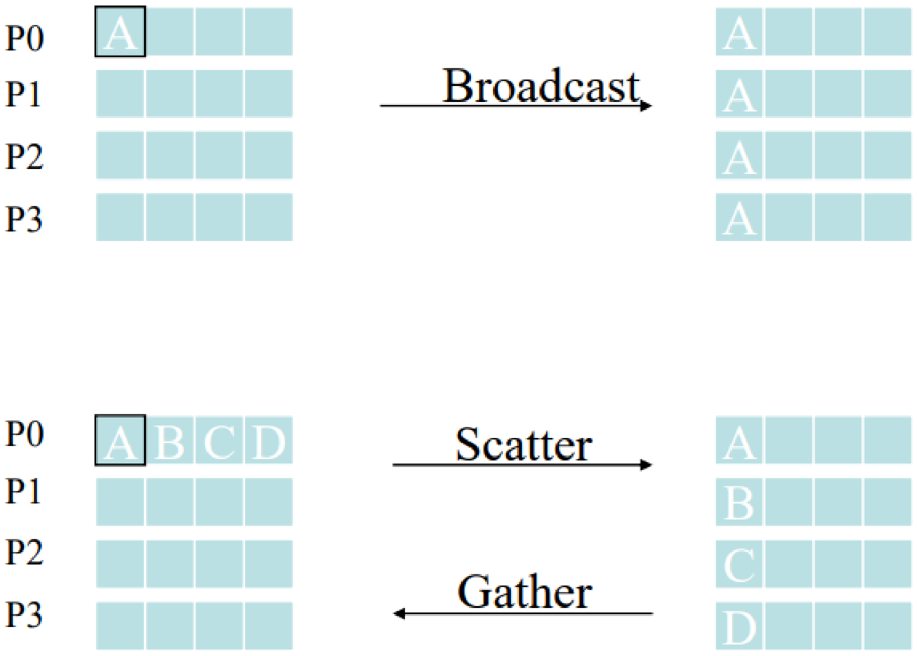
- Collective routines provide a higher-level way to organize a parallel program
  - Each process executes the same communication operations
  - Communication and computation is coordinated among a group of processes in a communicator
  - Tags are not used
- Three classes of operations
  - **synchronization**
  - **data movement**
  - **collective computation**

Synchronization

- int MPI_Barrier(MPI_Comm comm)
  - Blocks until all processes in the group of the communicator comm call it
- Not used often
- Sometime used in measuring performance and load balancing



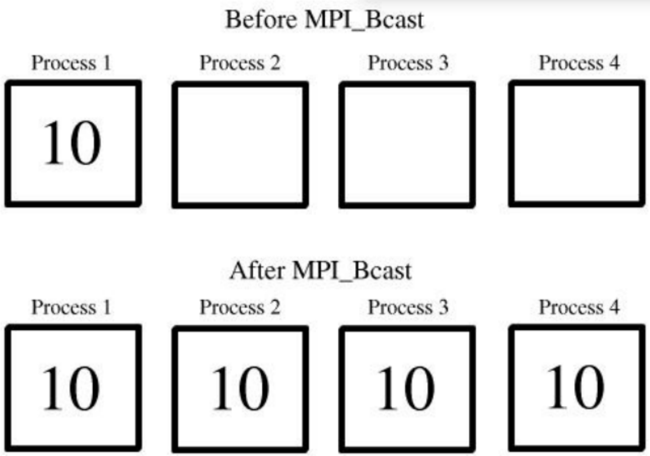
Collective Data Movement: Broadcast, Scatter and Gather



Broadcast

- Data belonging to a single process is <u>sent to all of the processes in the communicator</u>

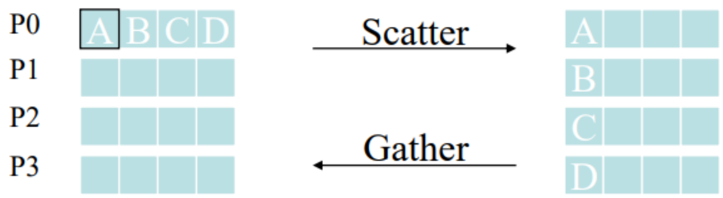
```
int MPI_Bcast(
        void*          data_p       /* in/out */,
        int            count        /* in     */,
        MPI_Datatype   datatype     /* in     */,
        int            source_proc  /* in     */,
        MPI_Comm       comm         /* in     */);
```



- All collective operations <u>must be called by all processes in the communicator</u>
- <u>MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast</u>
  - "source_proc" argument is the rank of the sender
  - MPI which process originates the broadcast and which receive

Scatter and Gather

- MPI_Scatter: can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes
- MPI_Gather: Collect all of the components of the vector onto process 0, and then process 0 can process all of the components




```
int MPI_Scatter(
      void*          send_buf_p   /* in   */,
      int            send_count   /* in   */,
      MPI_Datatype   send_type    /* in   */,
      void*          recv_buf_p   /* out  */,
      int            recv_count   /* in   */,
      MPI_Datatype   recv_type    /* in   */,
      int            src_proc     /* in   */,
      MPI_Comm       comm         /* in   */);
```

```
int MPI_Gather(
      void*          send_buf_p   /* in   */,
      int            send_count   /* in   */,
      MPI_Datatype   send_type    /* in   */,
      void*          recv_buf_p   /* out  */,
      int            recv_count   /* in   */,
      MPI_Datatype   recv_type    /* in   */,
      int            dest_proc    /* in   */,
      MPI_Comm       comm         /* in   */);
```

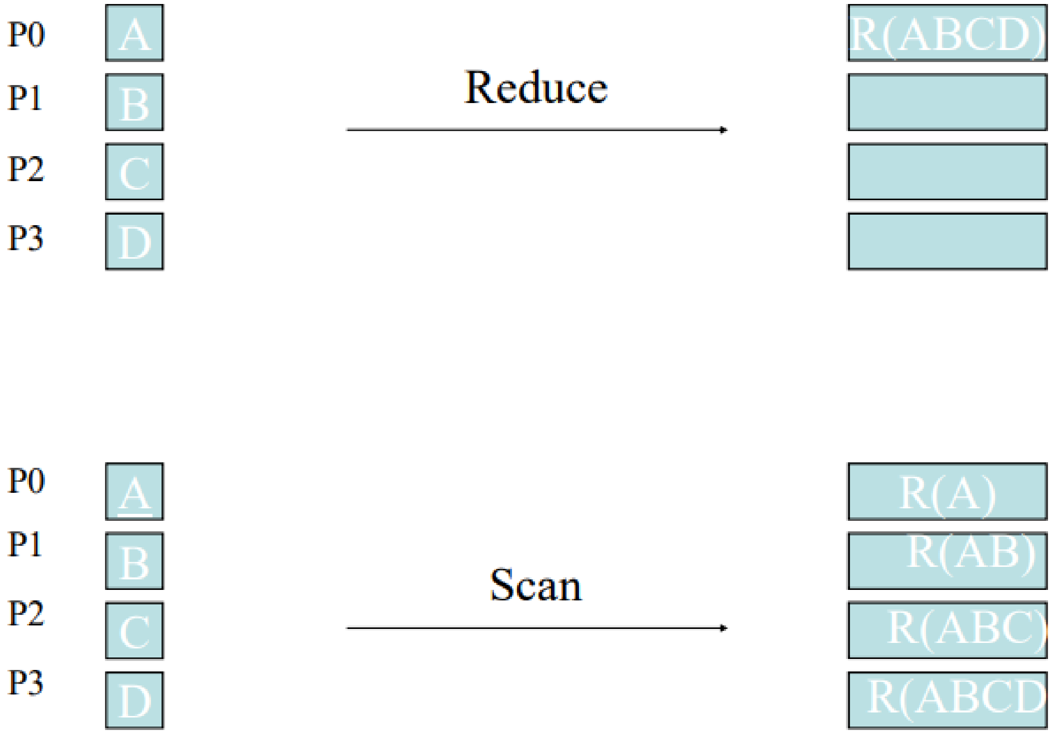
Example: A version of Get_input that uses MPI_Bcast

- Get_input() : get the input data and broadcast to other process
- process 0 as the root process originates the broadcast, other process receive the data: a_p , b_p , n_p

```
void Get_input(
        int      my_rank  /* in  */,
        int      comm_sz  /* in  */,
        double*  a_p      /* out */,
        double*  b_p      /* out */,
        int*     n_p      /* out */) {

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}  /* Get_input */
```

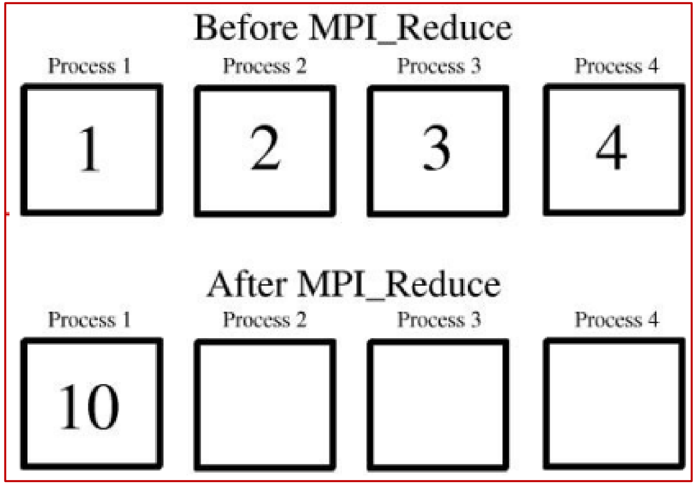
Collective Computation: Reduce,Scan



Reduce and Scan in MPI

- The all-to-one reduction operation is

```
int MPI_Reduce(
      void*          input_data_p   /* in   */,
      void*          output_data_p  /* out  */,
      int            count          /* in   */,
      MPI_Datatype   datatype       /* in   */,
      MPI_Op         operator       /* in   */,
      int            dest_process   /* in   */,
      MPI_Comm       comm           /* in   */);
```



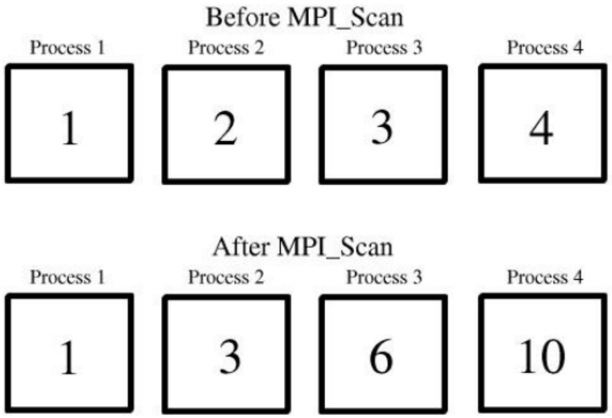

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

Reduce and Scan in MPI

- The scan operation is

```
int MPI_Scan(
        void *sendbuf,
        void *recvbuf,
        int count,
        MPI_Datatype datatype,
        MPI_Op op,
        MPI_Comm comm );
```



Predefined reduction operators in MPI

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Example of MPI PI program using 6 Functions

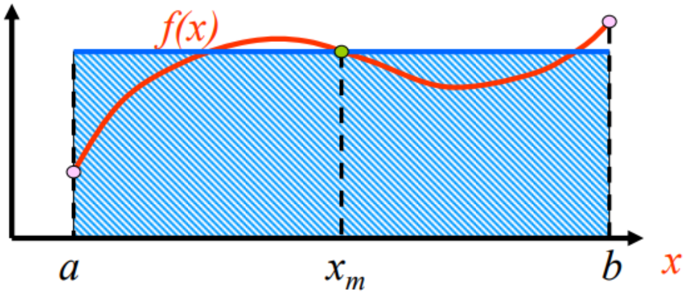
$$\pi = 4\int_0^1 \frac{1}{1+x^2}\,dx$$

- **Using basic MPI functions:**
  - **MPI_INIT**
  - **MPI_FINALIZE**
  - **MPI_COMM_SIZE**
  - **MPI_COMM_RANK**
- **Using MPI collectives:**
  - **MPI_BCAST**
  - **MPI_REDUCE**

Midpoint Rule for PI

$$\int_a^b f(x)dx \approx (b-a)f(x_m)$$

$$\pi = 4\int_0^1 \frac{1}{1+x^2}\,\mathrm{d}x$$



$$\int_{x=0}^{1} \frac{1}{1+x^2} \approx \sum_{i=1}^{n} \frac{1}{1+(\frac{i-0.5}{n})^2}$$

Example: PI in C with MPI-1

$$\int_{x=0}^{1} \frac{1}{1+x^2} \approx \sum_{i=1}^{n} \frac{1}{1+(\frac{i-0.5}{n})^2}$$

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

Input and broadcast parameters

Example: PI in C with MPI-2

$$\int_{x=0}^{1} \frac{1}{1+x^2} \approx \sum_{i=1}^{n} \frac{1}{1+(\frac{i-0.5}{n})^2}$$

```
    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;
}
```

Compute local pi values

Compute summation

Collective vs. Point-to-Point Communications

- All the processes in the communicator must call the same collective function
  - For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash

Wrong

```
if(my_rank==0)
MPI_Reduce(&a,&b,1,MPI_INT, MPI_SUM, 0,MPI_COMM_WORLD);
else
MPI_Recv(&a, MPI_INT, MPI_SUM,0,0, MPI_COMM_WORLD);
```

Collective vs. Point-to-Point Communications

- The arguments passed by each process to an MPI collective communication must be "compatible"
  - For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash

Wrong

```
if(my_rank==0)
MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
else
MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 1, MPI_COMM_WORLD);
```

parallel programming

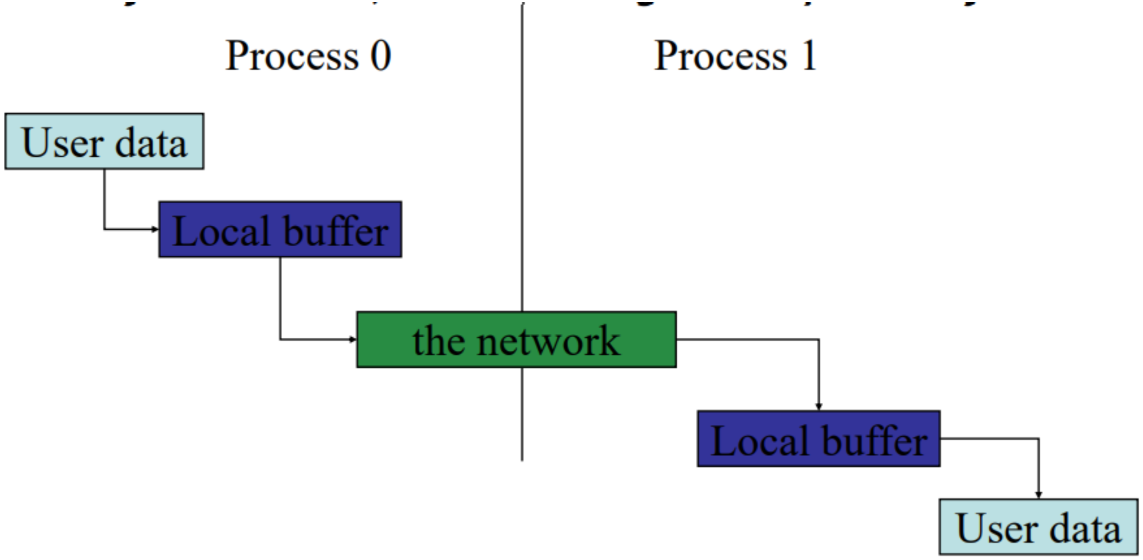
Safety Issues in MPI programs

Safety in MPI programs

- The MPI standard allows MPI_Send to behave in two different ways
  - it can simply copy the message into an MPI managed buffer and return
  - or it can block until the matching call to MPI_Recv starts

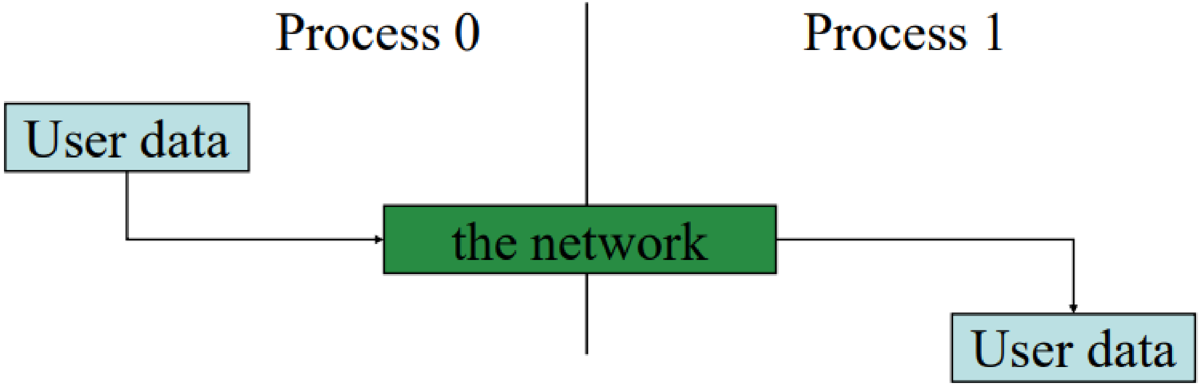
Buffer a message implicitly during MPI_Send()

When you send data, where does it go? One possibility is:



Avoiding Buffering

- Avoiding copies uses less memory
- May use more or less time



MPI_Send() waits until a matching receive is executed.

Safety in MPI programs

- Many implementations of MPI set a threshold at which the system switches from buffering to blocking
  - Relatively small messages will be buffered by MPI_Send
  - Larger messages, will cause it to block

- If the MPI_Send() executed by each process blocks, no process will be able to start executing a call to MPI_Recv, and the program will hang or deadlock
  - Each process is blocked waiting for an event that will never happen

Example of unsafe MPI code with possible 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 this code?

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 in which to store the data sent until it can be received

Safety in MPI programs

- A program that relies on MPI provided buffering is said to be unsafe
  - Such a program may run without problems for various sets of input, but it may hang or crash with other sets
- How can we tell if a program is unsafe
  - Replace MPI_Send() with MPI_Ssend()
    - ✓ The extra "s" stands for synchronous and MPI_Ssend is guaranteed to block until the matching receive starts
    - ✓ If the new program does not hang/crash, the original program is safe
    - ✓ MPI_Send() and MPI_Ssend() have the same arguments

```
int MPI_Ssend(
      void*          msg_buf_p      /* in  */,
      int            msg_size       /* in  */,
      MPI_Datatype   msg_type       /* in  */,
      int            dest           /* in  */,
      int            tag            /* in  */,
      MPI_Comm       communicator   /* in  */);
```

Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

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

- Simultaneous send and receive in one call:

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

- Use MPI_Sendrecv() to conduct a blocking send and a receive in a single call

```
int MPI_Sendrecv(
      void*          send_buf_p     /* in  */,
      int            send_buf_size  /* in  */,
      MPI_Datatype   send_buf_type  /* in  */,
      int            dest           /* in  */,
      int            send_tag       /* in  */,
      void*          recv_buf_p     /* out */,
      int            recv_buf_size  /* in  */,
      MPI_Datatype   recv_buf_type  /* in  */,
      int            source         /* in  */,
      int            recv_tag       /* in  */,
      MPI_Comm       communicator   /* in  */,
      MPI_Status*    status_p       /* in  */);
```

EuroMPI/USA 2022: 28th MPI Users' Group Meeting

EuroMPI/USA 2022 | **Home** | Call for Papers | Organizers | Submission | Program | Venue | Registration | Archive

WELCOME TO EUROMPI/USA 2022

In 2022, EuroMPI/USA Conference will take place in Chattanooga, Tennessee, USA at the University of Tennessee at Chattanooga on September 26-28, 2022. The conference will be co-located with the 18th International Workshop on OpenMP (IWOMP 2022) that will be held on September 27-30, 2022. The MPI Forum will also meet following the EuroMPI/USA Conference.

The EuroMPI/USA conference is the preeminent meeting for users, developers

Important Dates

Abstracts Submission Deadline: June 1, 2022 (AOE)

Full Paper Submission Deadline: June 8, 2022 (AOE)

Short Papers and Position Papers: June 20, 2022

Conclusion

- Message Passing Paradigm MPI
  - Message-Passing Programming Model
  - An overview of MPI programming
    - Six MPI functions and hello sample
    - How to compile/run
  - Send/Receive communication
    - Application Example: Parallelizing numerical integration with MPI
  - Collective group communication
    - Application Examples: Pi computation
  - Safety Issues in MPI programs