Introduction to MPI

Shaohao Chen

ORCD at MIT

Outline

- A brief overview of MPI
- Point-to-point communication
- Collective communication
- Derived datatype

MPI Overview

❑ Message Passing Interface (MPI) is a standard for parallel computing on a computer cluster.

❑ MPI is a library that includes C, C++, and Fortran routines.

❑ Computations are carried out simultaneously by multiple processes.

❑ Data is distributed to multiple processes.

❑ Data communication between processes is enabled by MPI subroutine/function calls.

✓ Typically, each process is mapped to one physical CPU core to achieve maximum performance.

❑ MPI implementations:

- OpenMPI
- MPICH, MVAPICH, Intel MPI

The first MPI program in C: Hello world!

- Hello world in C

```
#include <mpi.h>
main(int argc, char** argv){
    int my_rank, my_size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &my_size);
    printf("Hello from %d of %d.\n", my_rank, my_size);
    MPI_Finalize();
}
```

The first MPI program in Fortran: Hello world!

- Hello world in Fortran

```
program hello
    include 'mpif.h'
    integer my_rank, my_size, errcode
    call MPI_INIT(errcode)
    call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, errcode)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, my_size, errcode)
    print *, 'Hello from ', my_rank, 'of', my_size, '.'
    call MPI_FINALIZE(errcode)
end program hello
```

Basic Syntax

- ❑ Include the header file: mpi.h for C or mpif.h for Fortran

- ❑ MPI_INIT: This routine must be the first MPI routine you call (it does not have to be the first statement).
- ❑ MPI_FINALIZE: This is the companion to MPI_Init. It must be the last MPI call.
- ✓ MPI_INIT and MPI_FINALIZE appear in any MPI program.

- ❑ MPI_COMM_RANK: Returns the rank of the process. This is the only thing that sets each process apart from its companions.
- ❑ MPI_COMM_SIZE: Returns the total number of processes.
- ❑ MPI_COMM_WORLD: This is a communicator. Use MPI_COMM_WORLD unless you want to enable communication in complicated patterns.

- ❑ The error code is returned to the last argument in Fortran, while it is returned to the function value in C.

Compile and run MPI programs

Compile C/Fortran codes

```
mpicc name.c -o  name
mpif90 name.f90  -o  name
```

Run MPI programs

```
mpirun -np 4 ./name
```

Analysis of the output

```
$ mpirun -np 4 ./hello
Hello from 1 of 4.
Hello from 2 of 4.
Hello from 0 of 4.
Hello from 3 of 4.
```

❑ The MPI rank and size is printed by every process.

❑ Output is “disordered”. The output order is random.

❑ The output of all processes are printed on the session of the master process.

Basic MPI programming

- ❑ Point-to-point communication: MPI_Send, MPI_Recv
- ❑ Exercise: Circular shift and ring programs
- ❑ Collective communication: MPI_Bcast, MPI_Reduce
- ❑ Exercise: Compute the value of Pi
- ❑ Exercise: Parallelize Laplace solver using 1D decomposition

Point-to-point communication (1): Send

- ❑ One process sends a message to another process.

- ❑ Syntax:

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

- ✓ data: Initial address of send data.
- ✓ count: Number of elements send (nonnegative integer).
- ✓ datatype: Datatype of the send data.
- ✓ dest: Rank of destination(integer).
- ✓ tag: Message tag (integer).
- ✓ comm: Communicator.

Point-to-point communication (2): Receive

- ❑ One process receives a matching massage from another process.

- ❑ Syntax:

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

- ✓ data: Initial address of receive data.
- ✓ count: Maximum number of elements to receive (integer).
- ✓ datatype: Datatype of receive data.
- ✓ source: Rank of source (integer).
- ✓ tag: Message tag (integer).
- ✓ comm: Communicator (handle).
- ✓ status: Status object (status).

A C example: send and receive a number between two processes

```
int  my_rank, numbertoreceive, numbertosend;
MPI_Init(&argc, &argv);
MPI_Status status;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank==0){
   numbertosend=36;
   MPI_Send( &numbertosend, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
}
else if (my_rank==1){
   MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
   printf("Number received is: %d\n", numbertoreceive);
}
MPI_Finalize();
```

A Fortran example: send and receive a number between two processes

```
integer my_rank, numbertoreceive, numbertosend, errcode, status(MPI_STATUS_SIZE)
call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, errcode)
if (my_rank.EQ.0) then
   numbertosend = 36
   call MPI_Send( numbertosend, 1,MPI_INTEGER, 1, 10, MPI_COMM_WORLD, errcode)
elseif (my_rank.EQ.1) then
   call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, status, errcode)
   print *, 'Number received is:', numbertoreceive
endif
call MPI_FINALIZE(errcode)
```

Blocking Receives and Sends

❑ MPI_Recv is always blocking.

✓ Blocking means the function call will not return until the receive is completed.

✓ It is safe to use the received data right after calling MPI_Recv.

❑ MPI_Send try not to block, but don’t guarantee it.

✓ If the data size is not larger than that of the send buffer, MPI_Send is not blocking. The data is sent to the receive buffer without waiting.

✓ But if the data size is larger than that of the send buffer, MPI_Send is blocking. It first sends a chunk of data, then stops sending when the send buffer is full , and will restart sending when the send buffer becomes empty again.

✓ The later case often happens, so it is OK to think that MPI_Send is blocking.

A deadlock due to blocking receives

- ❑ An example: swapping arrays between two processes.
- ✓ The following code meets a deadlock situation and will hang forever.
- ✓ Both processes are blocked at MPI_Recv no matter how large the data size is.

```
int n=10;      // a small data size
int my_rank, n_send1[n], n_send2[n], n_recv1[n], n_recv2[n];
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank==0){
    MPI_Recv( &n_recv2, n, MPI_INT, 1, 11, MPI_COMM_WORLD, NULL);
    MPI_Send( &n_send1, n, MPI_INT, 1, 10, MPI_COMM_WORLD);
}
else if (my_rank==1){
    MPI_Recv( &n_recv1, n, MPI_INT, 0, 10, MPI_COMM_WORLD, NULL);
    MPI_Send( &n_send2, n, MPI_INT, 0, 11, MPI_COMM_WORLD);
}
```

A deadlock due to blocking sends

- ✓ If the sizes of the send arrays are large enough, MPI_Send becomes blocking, then the following code meets a deadlock situation.
- ✓ Both processes are blocked at MPI_Send for a large data size.

```
int n=5000;     // a large data size
int my_rank, n_send1[n], n_send2[n], n_recv1[n], n_recv2[n];
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank==0){
    MPI_Send( &n_send1, n, MPI_INT, 1, 10, MPI_COMM_WORLD);
    MPI_Recv( &n_recv2, n, MPI_INT, 1, 11, MPI_COMM_WORLD, NULL);
}
else if (my_rank==1){
    MPI_Send( &n_send2, n, MPI_INT, 0, 11, MPI_COMM_WORLD);
    MPI_Recv( &n_recv1, n, MPI_INT, 0, 10, MPI_COMM_WORLD, NULL);
}
```

Break the deadlock

✓ Send and receive are coordinated, so there is no deadlock.

```
int n=5000;       // a large data size
int my_rank, n_send1[n], n_send2[n], n_recv1[n], n_recv2[n];
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank==0){
   MPI_Send( &n_send1, n, MPI_INT, 1, 10, MPI_COMM_WORLD);
   MPI_Recv( &n_recv2, n, MPI_INT, 1, 11, MPI_COMM_WORLD, NULL);
}
else if (my_rank==1){
   MPI_Recv( &n_recv1, n, MPI_INT, 0, 10, MPI_COMM_WORLD, NULL);
   MPI_Send( &n_send2, n, MPI_INT, 0, 11, MPI_COMM_WORLD);
}
```

Exercise: Circular shift program

❑ Write an MPI program (in C or Fortran) to complete the following tasks.

Every process sends its rank to its right neighbor and receives the rank of its left neighbor. The process with the largest rank sends its rank to process 0.

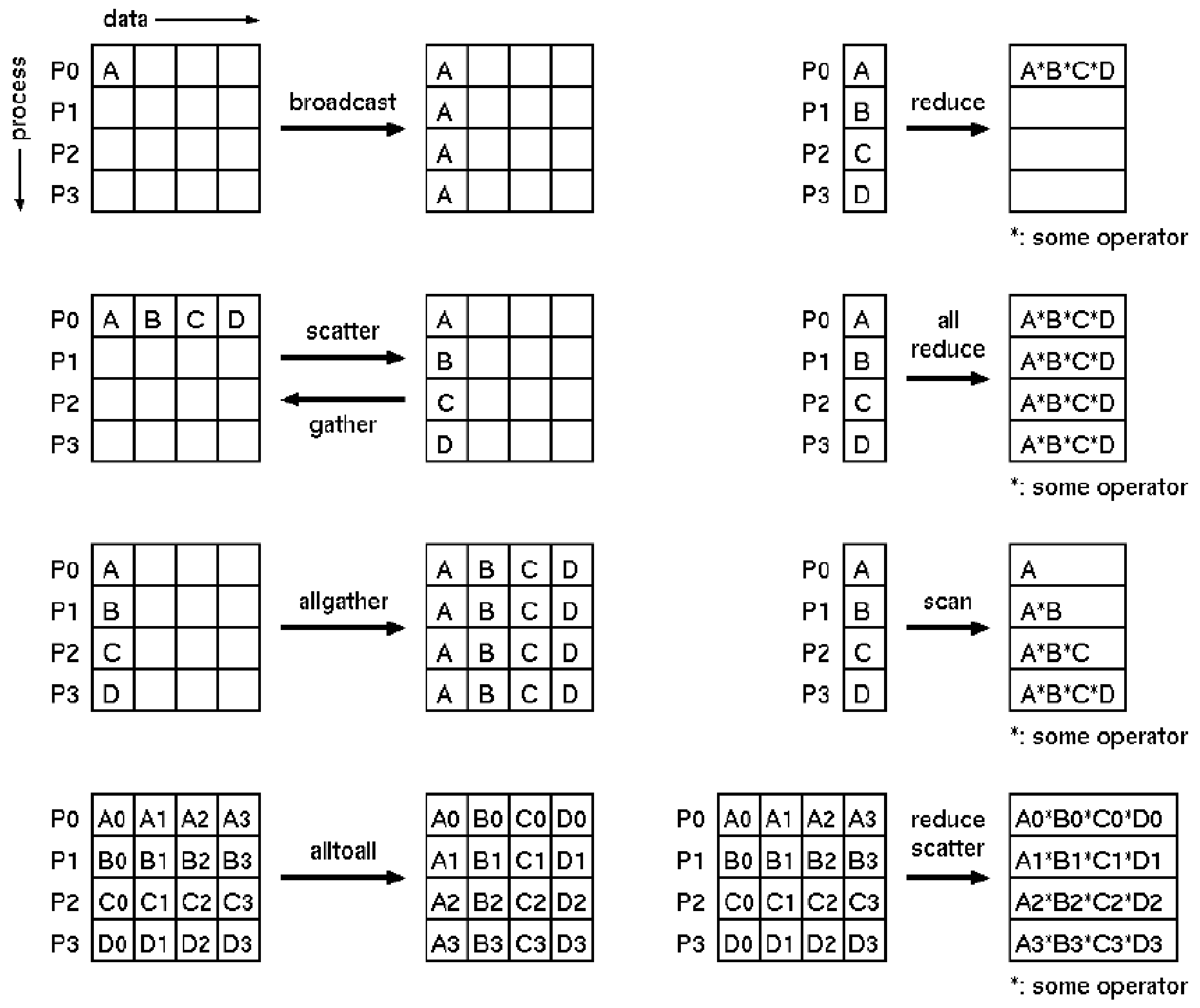
✓ Analysis:

1. Use MPI_Send and MPI_Recv (or MPI_Sendrecv).

2. Make sure every MPI_Send is associated with an MPI_Recv. Try to avoid deadlocks.

3. The data size is small, so MPI_Send is not blocking.

Collective Communication



- ❑ Collective:
- ✓ One to many
- ✓ Many to one
- ✓ Many to many

Reference:
Practical MPI Programming, IBM Redbook

Collective communication: Broadcast

❑ The root process broadcasts a massage to all other processes.

❑ Syntax:

int MPI_Bcast (void * data, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

✓ data: Initial address of the broadcast data.

✓ count: Number of elements the data (nonnegative integer).

✓ datatype: Datatype of the data.

✓ roort: Rank of the root process (integer).

✓ comm: Communicator (handle).

❑ The Bcast and Reduce routines are parallelized based on a binary-tree algorithm with O(log(N)) time complexity.

Collective communication: Reduce

- ❑ Reduce values of a variable on all processes to a single value and stores the value on the root process.

- ❑ Syntax:

```
int MPI_Reduce (const void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op,
int root, MPI_Comm comm)
```

- ✓ send_data: Initial address of the send data.
- ✓ recv_data: Initial address of the receive data.
- ✓ count: Number of elements the data (nonnegative integer).
- ✓ datatype: Datatype of the data.
- ✓ op: Reduction operation
- ✓ root: Rank of the root process (integer).
- ✓ comm: Communicator.

Reduction Operations

- MPI_MAX - Returns the maximum element.
- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical and across the elements.
- MPI_LOR - Performs a logical or across the elements.
- MPI_BAND - Performs a bitwise and across the bits of the elements.
- MPI_BOR - Performs a bitwise or across the bits of the elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it.
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

A C example for Bcast and Reduce

1. Broadcast the value of a variable x from process 0 to all other processes.
2. Multiply x by the MPI rank on all processes.
3. Compute the sum of all products and print it on process 0.

```
int my_rank, s=0, x=0;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if(my_rank==0) x=2;
MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
x *= my_rank;
MPI_Reduce(&x, &s, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if(my_rank==0) printf("The sum is %d.\n", s);
MPI_Finalize();
```

A Fortran example for Bcast and Reduce

1. Broadcast the value of a variable x from process 0 to all other processes.
2. Multiply x by the MPI rank on all processes.
3. Compute the sum of all products and print it on process 0.

```
integer errcode, my_rank, s, x
call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, errcode)
if(my_rank==0) x=2
call MPI_Bcast(x, 1, MPI_INT, 0, MPI_COMM_WORLD, errcode)
x = x * my_rank
call MPI_Reduce(x, s, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD, errcode)
if(my_rank==0) print *, 'The sum is:', s
call MPI_FINALIZE(errcode)
```

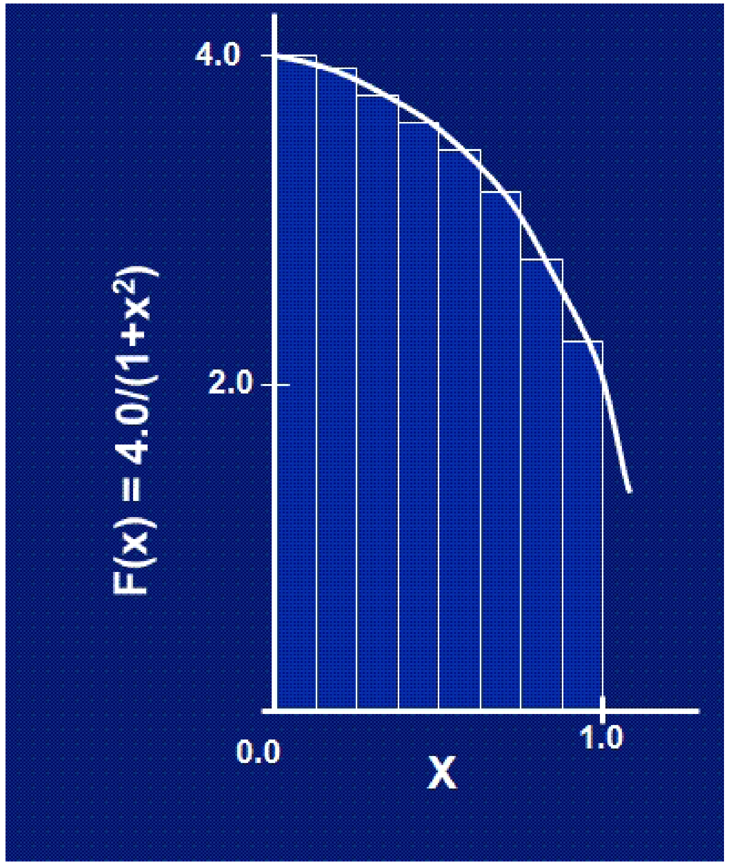
Exercise: Compute the value of Pi

- ❑ Provided a serial code that computes the value of Pi based on this integral formula,

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

parallelize the code using MPI.

- ✓ Hints: Distributes the grids to multiple processes. Each process performs its local integration. Use MPI_Bcast to broadcast the total number of grids. Use MPI_Reduce to obtain the total integration.



More on MPI

- More on collective communication:

  MPI_scatter, MPI_gather, MPI_Allreduce, MPI_Allgather , MPI_Alltoall
- Derived datatype: Contiguous, vector, indexed, and struct datatypes
- Exercise: Parallelize Laplace solver using 2D decomposition

Collective communication: Allreduce, Allgather

- MPI_Allreduce is the equivalent of doing MPI_Reduce followed by an MPI_Bcast. The root process obtains the reduced value and broadcasts it to all other processes.
- MPI_Allgather is the equivalent of doing MPI_Gather followed by an MPI_Bcast. The root process gathers the values and broadcasts them to all other processes.

- Syntax:

```
int MPI_Allreduce (const void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Allgather (const void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, MPI_Comm comm)
```

Quiz

❑ What is the result of the following code on 4 processes?

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank % 2 == 0) {  // Even
    MPI_Allreduce(&rank, &evensum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    if (rank == 0)   printf("evensum = %d\n", evensum);
} else {    // Odd
    MPI_Allreduce(&rank, &oddsum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    if (rank == 1)   printf("oddsum = %d\n", oddsum);
}
```

a) evensum=2   oddsum=4

b) evensum=6   oddsum=0

c) evensum=6   oddsum=6

d) evensum=0   oddsum=0

Derived Datatype

❑ Derived datatype: for users to define a new datatype that is derived from old datatype(s).

❑ Why derived datatype?

✓ Noncontiguous messages

✓ Convenience for programming

✓ Possible better performance and less data movements

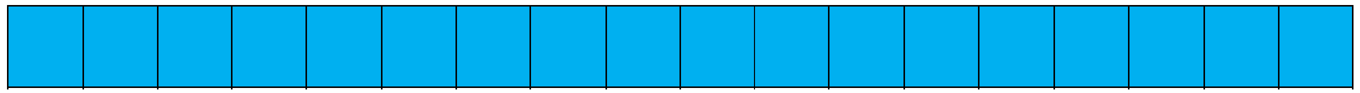
❑ Declare and commit a new datatype:

✓ MPI_Datatype *typename* : declare a new datatype

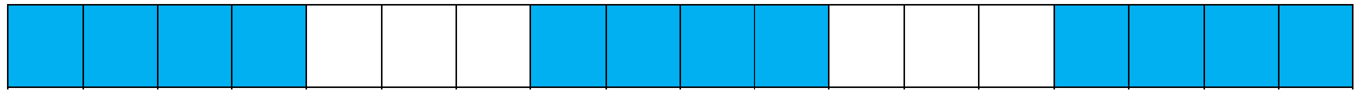
✓ MPI_Type_commit(&*typename*): commit the new datatype before using it.

Illustration of contiguous, vector, indexed and struct datatypes

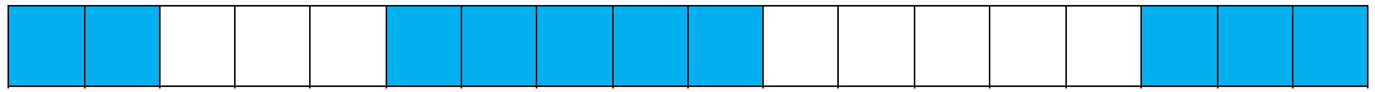
- Contiguous:



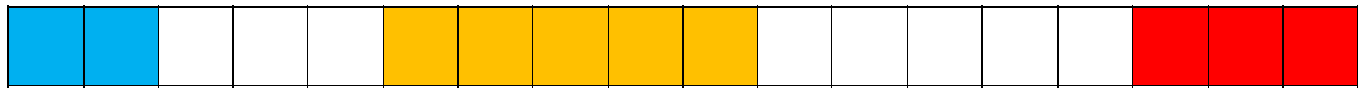
- Vector: non-contiguous, fixed block length and stride.



- Indexed: varying block lengths and strides.



- Struct: varying block lengths, strides and datatypes.



A C example for contiguous, vector and indexed datatypes

```
int n=18;
int blocklen[3] = {2, 5, 3 }, disp[3] = { 0, 5, 15 };
MPI_Datatype type1, type2, type3;
MPI_Type_contiguous(n, MPI_INT, &type1);     MPI_Type_commit(&type1);
MPI_Type_vector(3, 4, 7, MPI_INT, &type2);     MPI_Type_commit(&type2);
MPI_Type_indexed(3, blocklen, disp, MPI_INT, &type3);    MPI_Type_commit(&type3);
if (rank == 0){
        for (i=0; i<n; i++)   buffer[i] = i+1;
        MPI_Send(buffer, 1, type1, 1, 101, MPI_COMM_WORLD);
        MPI_Send(buffer, 1, type2, 1, 102, MPI_COMM_WORLD);
        MPI_Send(buffer, 1, type3, 1, 103, MPI_COMM_WORLD);
} else if (rank == 1) {
        MPI_Recv(buffer1, 1, type1, 0, 101, MPI_COMM_WORLD, &status);
        MPI_Recv(buffer2, 1, type2, 0, 102, MPI_COMM_WORLD, &status);
        MPI_Recv(buffer3, 1, type3, 0, 103, MPI_COMM_WORLD, &status);
}
```

What is not covered ……

- ❑ Non-blocking send and receive (overlapping computation and communication)
- ❑ Single-sided Communications
- ❑ Communicator and topology
- ❑ Remote Memory Access
- ❑ Hybrid Programming: MPI + OpenMP, MPI + OpenACC, MPI + CUDA, ……
- ❑ MPI-based libraries
- ❑ MPI I/O
- ❑ MPI with other languages: python, perl, R, ……

Exercise: Ring program

❑ Write an MPI code (in C or Fortran) to complete the following tasks.

Assign the value -1 to a variable named “token” on process 0, then pass the token around all processes in a ring-like fashion. The passing order is 0 → 1 → … → N → 0, where N is the maximum number of processes.

✓ Analysis:

1. Use MPI_Send and MPI_Recv (or MPI_Sendrecv).

2. Make sure every MPI_Send is associated with an MPI_Recv. Try to avoid deadlocks.

3. The data size is small, so MPI_Send is not blocking.

Exercise: Laplace Solver (version 1)

❑ Provided a serial code for solving the two-dimensional Laplace equation,

$$\nabla^2 f(x, y) = 0$$

parallelize the code using MPI.

✓ Analysis:

1. Decompose the grids into sub-grids. Divide the rows in C or divide the columns in Fortran. Each process owns one sub-grid.

2. Pass necessary data between sub-grids. (e.g. using MPI_Send and MPI_Recv). Be careful to avoid deadlocks.

3. Pass shared data between the root process and all other processes (e.g. use MPI_Bcast and MPI_Reduce).

Exercise: Laplace Solver (version 2)

❑ Rewrite an MPI program to solve the Laplace equation based on 2D decomposition.

✓ Analysis:

1. Decompose the grids into sub-grids. Divide both rows and columns. Each process owns one sub-grid.
2. Define necessary derived datatypes (e.g. MPI_contiguous and MPI_vector).
3. Pass necessary data between processes. (e.g. use MPI_Send and MPI_Recv). Be careful to avoid dead locks.
4. Pass shared data between the root process and all other processes (e.g. use MPI_Bcast and MPI_Reduce).

References

- *Practical MPI Programming, IBM Redbook, by Yukiya Aoyama and Jun Nakano*
- *Using MPI, Third Edition, by William Gropp, Ewing Lusk and Anthony Skjellum, The MIT Press*
- *Using Advanced MPI, by William Gropp, Torsten Hoefler, Rajeev Thakur and Ewing Lusk, The MIT Press*