Introduction to MPI

Shaohao Chen

Research Computing Services

Information Services and Technology

Boston University

Outline

- Brief overview on parallel computing and MPI
- Using MPI on BU SCC
- Basic MPI programming
- Advanced MPI programming

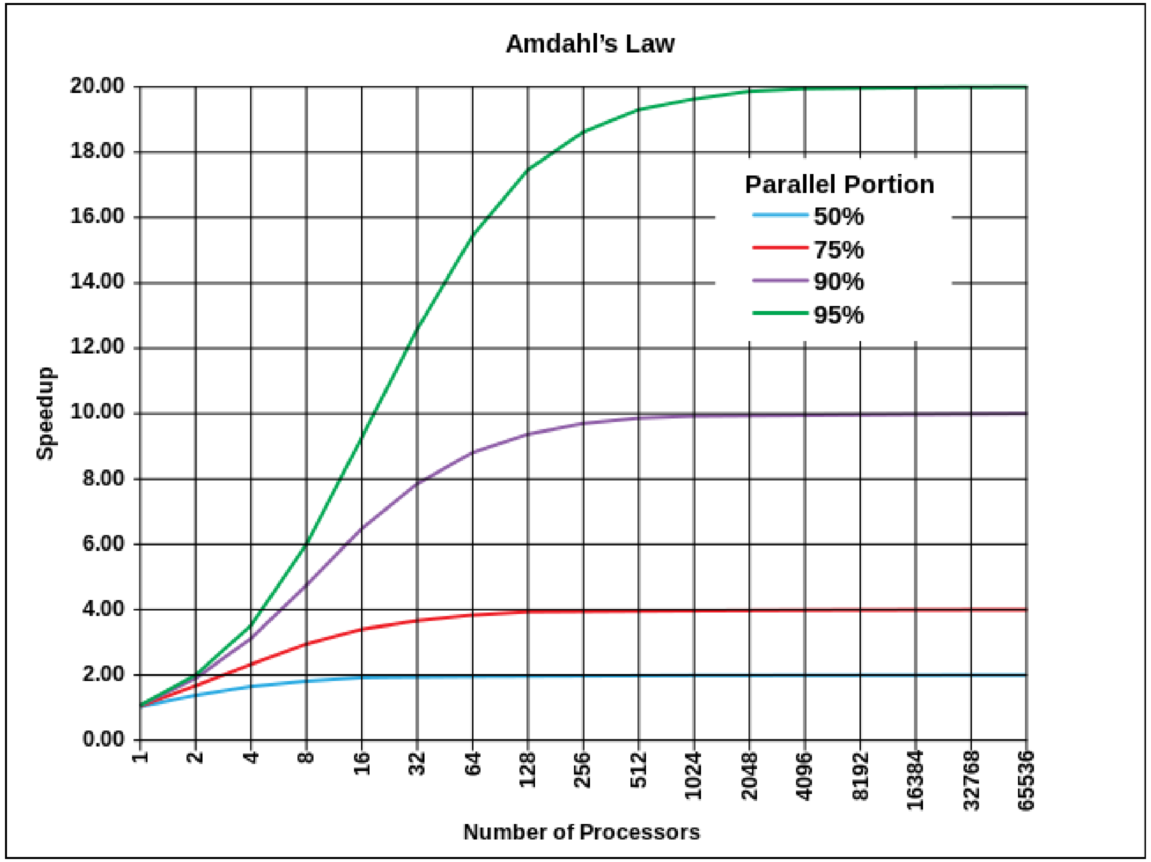
Parallel Computing

- ❑ Parallel computing is a type of computation in which many calculations are carried out simultaneously, based on the principle that large problems can often be divided into smaller ones, which are then solved at the same time.
- ❑ Speedup of a parallel program,

$$S(p) = {}^{T(1)}\!/_{T(p)} = \frac{1}{\alpha + {}^{1}\!/_{p}\,(1-\alpha)}$$

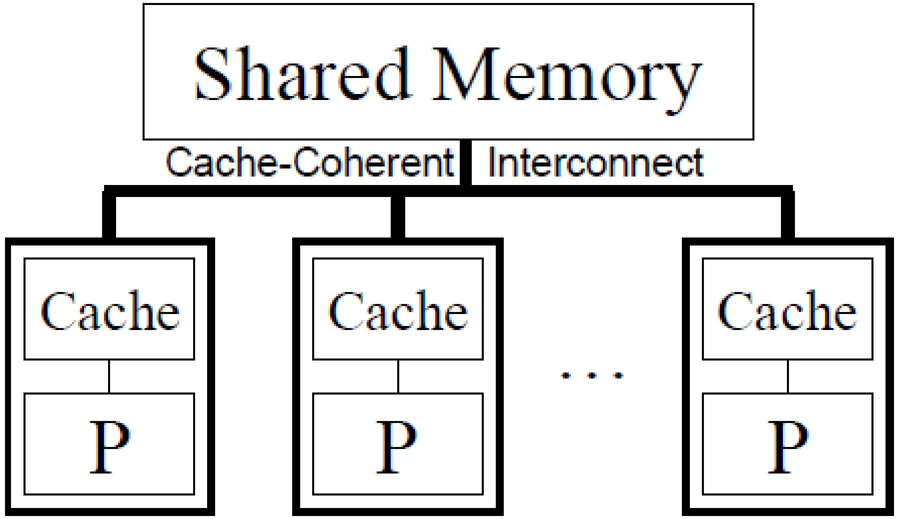
p: number of processors/cores,

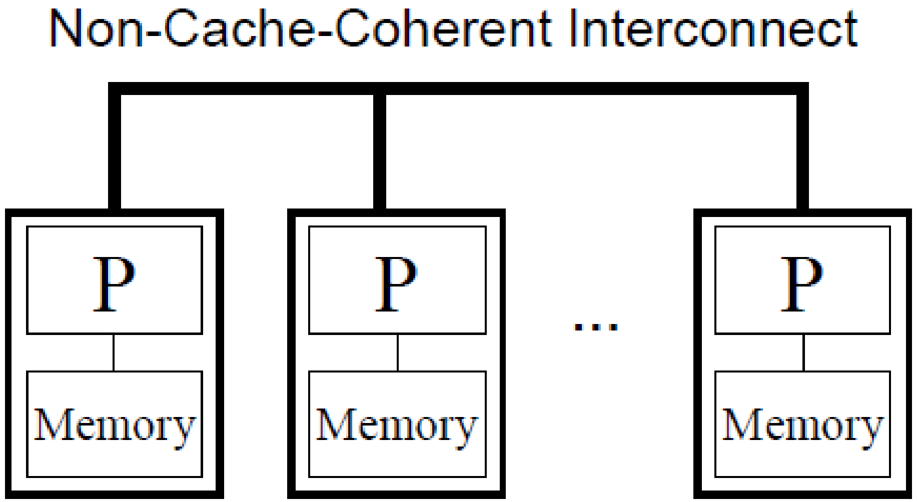
α: fraction of the program that is serial.



- The picture is from: https://en.wikipedia.org/wiki/Parallel_computing

Distributed and shared memory systems





- Shared memory system
- For example, a single node on a cluster
- Open Multi-processing (OpenMP) or MPI

- Distributed memory system
- For example, mutli nodes on a cluster
- Message Passing Interface (MPI)

✓ Figures are from the book *Using OpenMP: Portable Shared Memory Parallel Programming*

MPI Overview

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

❑ MPI is a Library. Provides library routines in C, C++, and Fortran.

❑ Computations are carried out simultaneously by multiple processes.

❑ Data is distributed to multiple processes. There is no shared data.

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

✓ Typically each process is mapped to one physical processor 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 code.

❑ 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 MPI codes on BU SCC

❑ Use GNU compiler (default) and OpenMPI

```
$ export MPI_COMPILER=gnu
$ mpicc name.c -o name
$ mpif90 name.f90 -o name
```

❑Use Portland Gourp Inc. (PGI) compiler and OpenMPI

```
$ export MPI_COMPILER=pgi
$ mpicc name.c -o name
$ mpif90 name.f90 -o name
```

Compile MPI codes on BU SCC (continued)

❑ Use Intel compiler and OpenMPI

```
$ module load openmpi/1.10.1_intel2016
$ mpicc name.c –o name
$ mpifort name.f90 –o name
```

❑ Check what compiler and MPI implementation are in use

```
$ mpicc --show
$ mpif90 --show
```

❑ For more information: http://www.bu.edu/tech/support/research/software-and-programming/programming/multiprocessor/#MPI

Interactive MPI jobs on BU SCC

❑ Request an interactive session with two 12-core nodes,

```
$ qlogin -pe mpi_12_tasks_per_node 24
```

❑ Check which nodes and cores are requested,

```
$ qstat -u userID -t
```

❑ Run an MPI executable

```
$ mpirun -np $NSLOTS ./executable
```

✓ Note: NSLOTS, representing the total number of requested CPU core, is an environmental variable provided by the job scheduler.

❑ Check whether the job really runs in parallel

```
$ top
```

Submit a batch MPI job on BU SCC

❑ Submit a batch job

```
$ qsub job.sh
```

❑ A typical job script is as following

```
#!/bin/bash
#$ -pe mpi_16_tasks_per_node 32
#$ -l h_rt=01:30:00
#$ -N job_name
mpirun -np $NSLOTS ./executable
```

✓ Note: No need to provide host file explicitly. The job scheduler automatically distributes MPI processes to the requested resources.

Exercise 1: hello world

1) Write an MPI hello-world code in either C or Fortran. Print the MPI ranks and size on all processes.
2) Compile the hello-world code.
3) Run the MPI hello-world program either in an interactive session or by submitting a batch job.

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 output by every process.

❑Output is “disordered”. The output order is random. It depends on which process finishes its work first.

❑ In a run on multiple nodes, the output of all nodes are printed on the session of the master node. This indicates implicit dada communication behind the scene.

Basic MPI programming

- ❑ Point-to-point communication: MPI_Send, MPI_Recv
- ❑ Exercise: Circular shift and ring programs
- ❑ Synchronization: MPI_Barrier
- ❑ 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

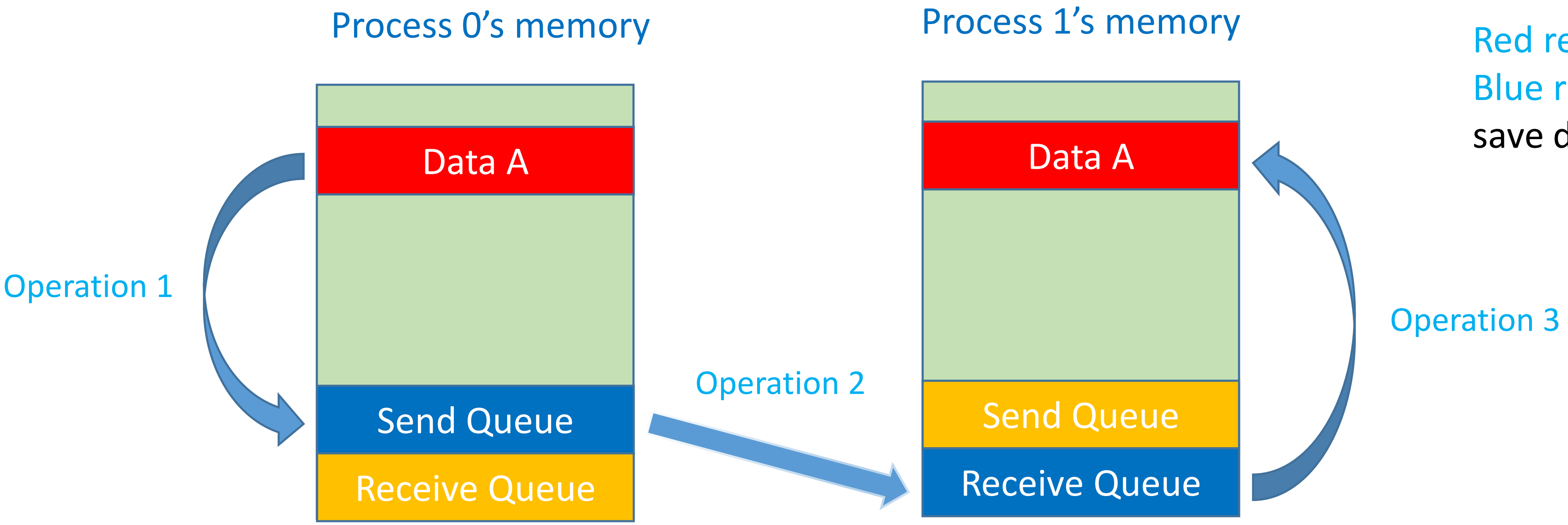
```
integer my_rank, numbertoreceive, numbertosend, errcode, status(MPI_STATUS_SIZE)
call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, errcode)
if (my_rank.EQ.0) then
    numbertosend = 36
    call MPI_Send( numbertosend, 1,MPI_INTEGER, 1, 10, MPI_COMM_WORLD, errcode)
elseif (my_rank.EQ.1) 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)
```

What actually happened behind the scene?

Operation 1: On process 0, MPI_Send copies the data to Send Queue/Buffer.

Operation 2: MPI_Send moves the data from process 0’s Send Queue to process 1’s Receive Queue/Buffer. (The rank of the destination is an input argument of MPI_Send, so it knows where the data should go to.)

Operation 3: On process 1, MPI_Recv checks whether the matching data has arrived (Source and tag are checked. But data type and counts are not checked). If not arrive, it waits until the matching data arrives. If arrives, it moves the data from the Receive Queue to process 1’s memory. This mechanism guarantees that the send data will not be “missed”.



Red regions: save data A.
Blue regions: temporarily save data A.

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 size of send data is smaller than that of the Send Queue, MPI_Send is not blocking --- the data is sent to the Receive Queue without waiting.

✓ But if the size of send data is larger than that of the Send Queue, MPI_Send is blocking --- it first sends a chunk of data, then stops sending when the Send Queue is full and will restart sending when the Send Queue becomes empty again (for example when the chunk of data has been moved to the Receive Queue).

✓ 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 (1)

✓ 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);
}
```

Point-to-point communication (3): Sendrecv

- ❑ The send-receive operations combine the sending of a message to one destination and the receiving of another message in one call.
- ❑ MPI_Sendrecv executes a blocking send and receive operation.

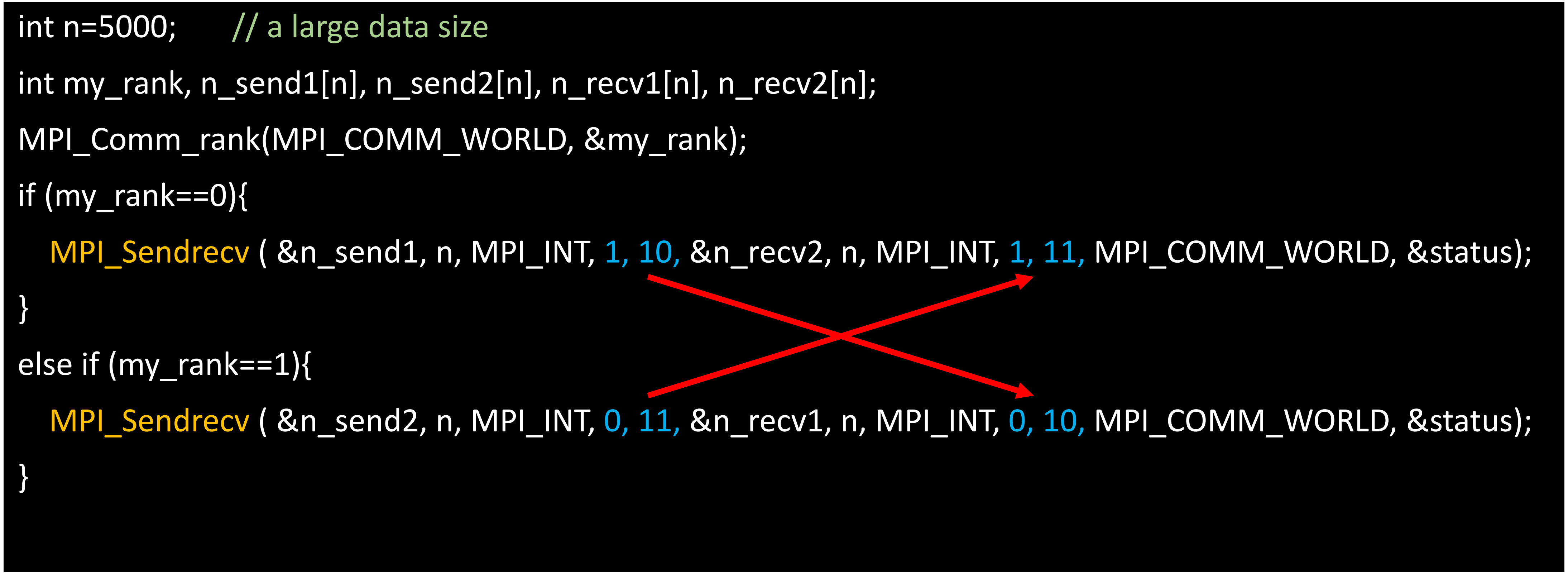
- ❑ Syntax:

int MPI_Sendrecv (const void* senddata, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void* recvdata, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status* status)

- ✓ The communicator for send and receive must be the same.
- ✓ The destination for send and the source for receive may be the same or different.
- ✓ The tags, count, datatype for send and receive may be the same or different.
- ✓ The send buffer and the receive buffer must be disjoint.

Break the deadlock (2)

✓ Send and receive are automatically coordinated in MPI_Sendrecv, 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_Sendrecv ( &n_send1, n, MPI_INT, 1, 10, &n_recv2, n, MPI_INT, 1, 11, MPI_COMM_WORLD, &status);
}
else if (my_rank==1){
   MPI_Sendrecv ( &n_send2, n, MPI_INT, 0, 11, &n_recv1, n, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
}
```

Exercise 2: Circular shift and ring programs

❑ Write two MPI codes (in C or Fortran) to do the following tasks respectively:

1) Circular shift program: Every process sends its rank to its right neighbor and receives the rank of its left neighbor. (The process with the largest rank send its rank to process 0.)

2) Ring program: 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.

✓ Hints: Use MPI_Send and MPI_Recv (or MPI_Sendrecv). Make sure every MPI_Send corresponds to a matching MPI_Recv. Be careful to avoid deadlocks.

Synchronization: Barrier

❑ Blocks until all processes in the communicator have reached this routine.

❑ Syntax:

```
int MPI_Barrier (MPI_Comm comm)
```

✓ comm: Communicator.

Print in order

- ❑ Print in order using MPI_Barrier.

```
int my_rank, my_size, i;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &my_size);
for(i=0; i<my_size; i++){
  if(i==my_rank) printf("Hello from %d.\n", my_rank);
  MPI_Barrier(MPI_COMM_WORLD);
}
MPI_Finalize();
```

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).

❑ Broadcast can be also enabled by using MPI_Send and MPI_Recv. But MPI_Bcast is more efficient, because advanced algorithms (such as a binary-tree algorithm) are implemented in it.

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 MPI_Bcast and MPI_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.

```c
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 MPI_Bcast and MPI_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)
```

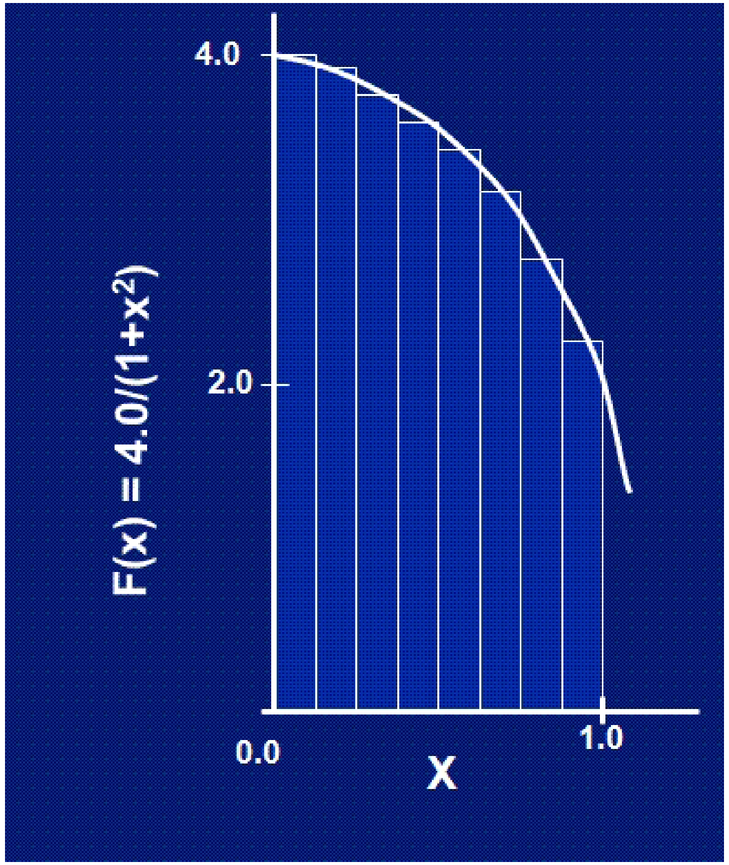
Exercies 3: Compute the value of Pi

- ❑ Provided a serial code (in C or Fortran) that computes the value of Pi based on the 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.



Exercise 4: Laplace Solver (version 1)

❑ Provided a serial code (in C or Fortran) for solving the two-dimensional Laplace equation,

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

parallelize the code using MPI.

✓ Analysis (see the slides Laplace Exercise for details):

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 dead locks.

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

Advanced MPI programming

- ❑ Non-blocking Sends and Receives:

  MPI_Isend, MPI_Irecv, MPI_Wait

- ❑ More on collective communication:

  MPI_scatter, MPI_gather, MPI_Allreduce, MPI_Allgather , MPI_Alltoall

- ❑ Quiz: Understanding MPI_Allreduce.
- ❑ Derived datatype: Coniguous, vector, indexed and struct datatypes
- ❑ Exercise: Parallelize Laplace solver using 2D decomposition

Non-blocking Send and Receive: Isend, Irecv

❑ MPI_Isend and MPI_Irecv perform non-blocking send and receive respectively, meaning that the function calls return before the communication is completed.

❑ MPI_Wait waits for an MPI request to complete.

❑ Syntax:

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

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

int MPI_Wait (MPI_Request* request, MPI_Status* status)

✓ request: communication request

Break the deadlock using non-blocking send and receive

✓ Both MPI_Isend and MPI_Irecv are non-blocking, so there is no deadlock in the following code.

✓ The performance of using non-blocking send and receive is better than that of using blocking send and receive. But be careful about the data safety.

```
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);
MPI_Request send_request, recv_request;
if (my_rank==0){
   MPI_Isend( &n_send1, n, MPI_INT, 1, 10, MPI_COMM_WORLD, &send_request);
   MPI_Irecv( &n_recv2, n, MPI_INT, 1, 11, MPI_COMM_WORLD, &recv_request);
}
else if (my_rank==1){
   MPI_Isend( &n_send2, n, MPI_INT, 0, 11, MPI_COMM_WORLD, &send_request);
   MPI_Irecv( &n_recv1, n, MPI_INT, 0, 10, MPI_COMM_WORLD, &recv_request);
}
```

Data safety for non-blocking sends and receives

- It is not safe to modify the send data right after calling MPI_Isend or to use the receive data right after calling MPI_Irecv.
- Use MPI_Wait to make sure the non-blocking send or receive is completed.
- If the two MPI_Wait functions were not called in the following code, the send/receive data would be modified/printed before the send/receive is completed.

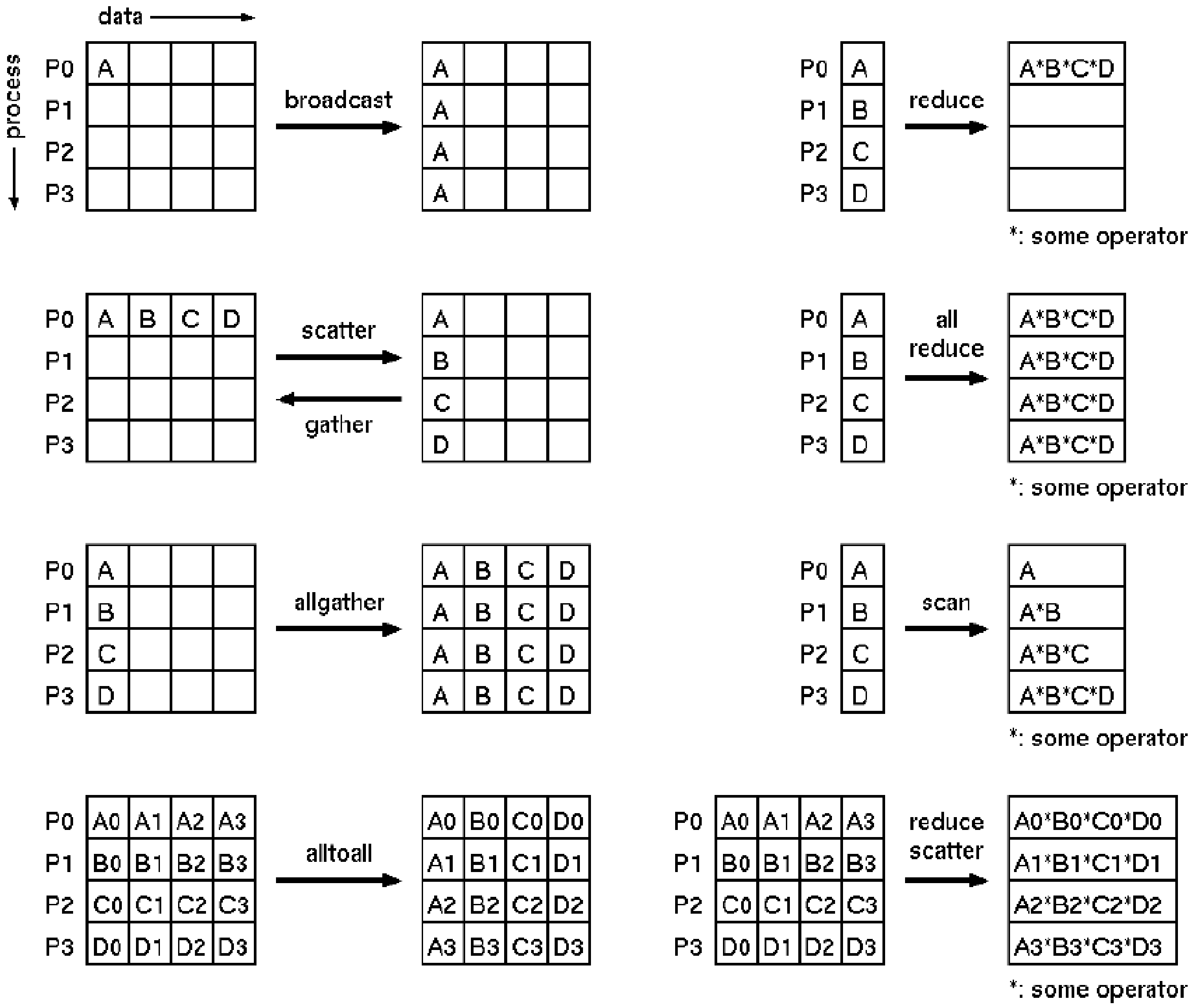
```
// Continued from the previous page
MPI_Wait (&send_request, NULL);
n_send1[4999]=101;
n_send2[4999]=201;
MPI_Wait (&recv_request, NULL);
if(my_rank==0) printf(" The last element on rank %d is %d. \n", my_rank, n_recv2[4999]);
if(my_rank==1) printf(" The last element on rank %d is %d. \n", my_rank, n_recv1[4999]);
```

Mixing blocking and non-blocking sends and receives

- ✓ MPI_Isend is non-blocking, so there is no deadlock in the following code.
- ✓ MPI_Recv is blocking, so the data is save to be used right after MPI_Recv .

```
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);
MPI_Request send_request;
if (my_rank==0){
   MPI_Isend( &n_send1, n, MPI_INT, 1, 10, MPI_COMM_WORLD, &send_request);
   MPI_Recv( &n_recv2, n, MPI_INT, 1, 11, MPI_COMM_WORLD);
}
else if (my_rank==1){
   MPI_Isend( &n_send2, n, MPI_INT, 0, 11, MPI_COMM_WORLD, &send_request);
   MPI_Recv( &n_recv1, n, MPI_INT, 0, 10, MPI_COMM_WORLD);
}
```

Collective Communication Subroutines



data →

process ↓

broadcast →

P0	A					A			
P1						A			
P2						A			
P3						A			

reduce →

P0	A	A*B*C*D
P1	B	
P2	C	
P3	D	

*: some operator

scatter → / ← gather

P0	A	B	C	D		A			
P1						B			
P2						C			
P3						D			

all reduce →

P0	A	A*B*C*D
P1	B	A*B*C*D
P2	C	A*B*C*D
P3	D	A*B*C*D

*: some operator

allgather →

P0	A					A	B	C	D
P1	B					A	B	C	D
P2	C					A	B	C	D
P3	D					A	B	C	D

scan →

P0	A	A
P1	B	A*B
P2	C	A*B*C
P3	D	A*B*C*D

*: some operator

alltoall →

P0	A0	A1	A2	A3		A0	B0	C0	D0
P1	B0	B1	B2	B3		A1	B1	C1	D1
P2	C0	C1	C2	C3		A2	B2	C2	D2
P3	D0	D1	D2	D3		A3	B3	C3	D3

reduce scatter →

P0	A0	A1	A2	A3		A0*B0*C0*D0
P1	B0	B1	B2	B3		A1*B1*C1*D1
P2	C0	C1	C2	C3		A2*B2*C2*D2
P3	D0	D1	D2	D3		A3*B3*C3*D3

*: some operator

❑ Collective:

✓ One to many

✓ Many to one

✓ Many to many

The picture is from:
Practical MPI Programming, IBM Redbook

Collective communication: MPI_Scatter

❑ The root process sends chunks of an array to all processes. Each non-root process receives a chunk of the array and stores it in receive buffer. The root process also copies a chunk of the array to its own receive buffer.

❑ Syntax:

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

✓ send_data: The send array that originally resides on the root process.

✓ send_count: Number of elements to be sent to each process (i.e. the chunk size). It is often (approximately) equal to the size of the array divided by the number of processes.

✓ send_datatype: Datatype of the send data.

✓ recv_data: The receive buffer on all processes.

✓ recv_count: Number of elements that the receive buffer can hold (i.e. the chunk size). It should be equal to send_count if send_datatype and recv_datatype are the same.

✓ recv_datatype: Datatype of the receive data.

✓ root: The rank of the root process.

Collective communication: MPI_Gather

- ❑ MPI_Gather is the inverse of MPI_Scatter.
- ❑ Each non-root process sends a chunk of data to the root process. The root process receives chunks of data and stores them (including its own chunk) in the receive buffer in the order of MPI ranks.

- ❑ Syntax:

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

- ✓ send_data: The send data on each process.
- ✓ send_count: Number of elements of the send data (i.e. the chunk size).
- ✓ send_datatype: Datatype of the send data.
- ✓ recv_data: The receive buffer on the root process.
- ✓ recv_count: Number of elements of the receive data (i.e. the chunk size, not the size of the receive buffer) . It should be equal to send_count if send_datatype and recv_datatype are the same.
- ✓ recv_datatype: Datatype of the receive data.
- ✓ root: The rank of the root process.

An example for MPI_Scatter and MPI_Gather

- ❑ Compute the average of all elements in an array.

```
int rank, nproc, i, m, n=100;
double sub_avg=0., global_avg=0.;
double * array = NULL, * sub_avgs = NULL;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
m = (int) n/nproc;        // chunk size
if(rank==0){   array = (double *) malloc(n*sizeof(double));
               for(i=0; i<n; i++)  array[i]=(double) i;     }

double * chunk = (double *) malloc(m*sizeof(double));
MPI_Scatter(array, m, MPI_DOUBLE, chunk, m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
for(i=0; i<m; i++) sub_avg += chunk[i];
sub_avg = sub_avg/(double)m;
MPI_Barrier(MPI_COMM_WORLD);

if(rank==0) sub_avgs = (double *) malloc(nproc*sizeof(double));
MPI_Gather(&sub_avg, 1, MPI_DOUBLE, sub_avgs, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if(rank==0){  for(i=0; i<nproc; i++)    global_avg += sub_avgs[i];    }
printf("The global average is: %f\n", global_avg/(double)nproc);

free(chunk);    if(rank==0){ free(array);  free(sub_avgs); }
```

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?

✓ Hints: Break down the code using MPI_Send and MPI_Recv, then analyze how the program steps forward.

```
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

Basic Datatypes

❑ Basic datatype for C:

✓ MPI_CHAR --- Signed char

✓ MPI_SHORT --- Signed short int

✓ MPI_INT --- Signed int

✓ MPI_LONG --- Signed long int

✓ MPI_UNSIGNED_CHAR --- Unsigned char

✓ MPI_UNSIGNED_SHORT --- Unsigned short

✓ MPI_UNSIGNED --- Unsigned int

✓ MPI_UNSIGNED_LONG --- Unsigned long int

✓ MPI_FLOAT --- Float

✓ MPI_DOUBLE --- Double

✓ MPI_LONG_DOUBLE --- Long double

❑ Basic datatype for Fortran:

✓ MPI_INTEGER --- INTEGER

✓ MPI_REAL --- REAL

✓ MPI_REAL8 --- REAL*8

✓ MPI_DOUBLE_PRECISION --- DOUBLE PRECISION

✓ MPI_COMPLEX --- COMPLEX

✓ MPI_LOGICAL --- LOGICAL

✓ MPI_CHARACTER --- CHARACTER(1)

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

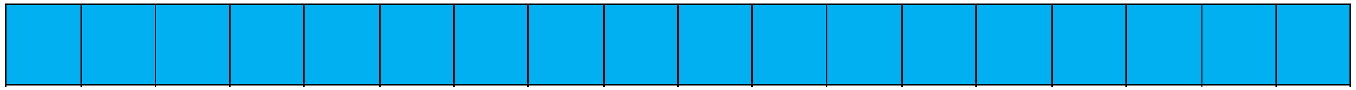
❑ Declare and commit a new datatype:

✓ MPI_Datatype *typename* : declare a new datatype

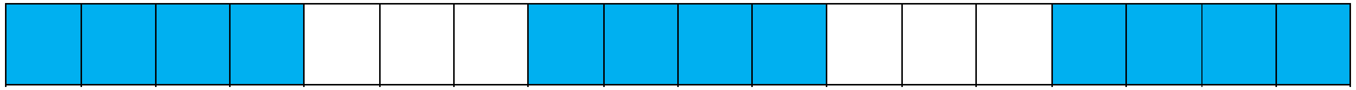
✓ MPI_Type_commit(&*typename*): commit the new datatype before use it.

Illustration of contiguous, vector, indexed and struct datatypes

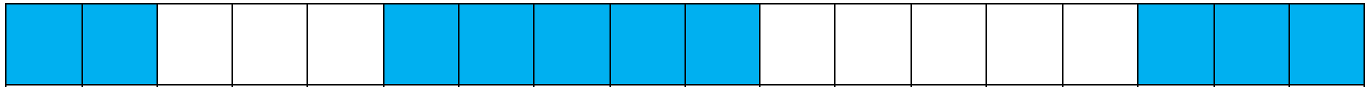
- Contiguous:



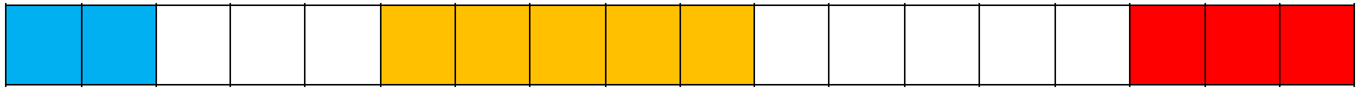
- Vector: non-contiguous, fixed block length and stride.



- Indexed: different block lengths and strides.



- Struct: different block lengths, strides and old datatypes.



Contiguous datatype

- ❑ Allows replication of an old data type into contiguous locations.
- ❑ For contiguous data.

❑ Syntax:

int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype * newtype)

- ✓ count: replication count (nonnegative integer)
- ✓ oldtype: old data type
- ✓ newtype: new data type

Vector datatype

- ❑ Allows replication of an old datatype into locations that consist of equally spaced blocks.
- ❑ Each block is a concatenation of the old datatype.
- ❑ The block length and the stride are fixed.

❑ Syntax:

int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype * newtype)

- ✓ count: number of blocks (nonnegative integer)
- ✓ blocklength: number of elements in each block (nonnegative integer)
- ✓ stride: number of elements between start of each block (integer)
- ✓ oldtype: old data type
- ✓ newtype: new data type

Indexed datatype

❑ Allows replication of an old datatype into a sequence of blocks.

❑ The block lengths and the strides may be different.

❑ Syntax:

int MPI_Type_indexed (int count, const int * blocklength, const int * displacements, MPI_Datatype oldtype, MPI_Datatype * newtype)

✓ count: number of blocks (nonnegative integer)

✓ blocklength: number of elements in each block (array of nonnegative integer)

✓ displacements: displacement of each block in multiples of oldtype (array of integer)

✓ oldtype: old data type

✓ newtype: new data type

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);
}
```

Struct datatype

- ❑ Allows each block to consist of replications of different datatypes.
- ❑ The block lengths, the strides and the old datatypes may be different.
- ❑ Give users full control to pack data.

- ❑ Syntax:

int MPI_Type_struct (int count, const int * blocklength, const MPI_Aint * displacements, MPI_Datatype oldtype, MPI_Datatype * newtype)

- ✓ count: number of blocks (nonnegative integer)
- ✓ blocklength: number of elements in each block (array of nonnegative integer)
- ✓ displacements: displacement of each block in multiples of bytes (array of integer)
- ✓ oldtype: old data type
- ✓ newtype: new data type

Pack size

❑ Returns the upper bound on the amount of space needed to pack a message.

❑ Syntax:

int MPI_Pack_size (int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

✓ count: Count argument to packing call (integer)

✓ datatype: Datatype argument to packing call

✓ comm: Communicator

✓ size: Upper bound on size of packed message, in unit of bytes (integer)

A C example for struct datatype

```
int psize;
int blocklens[3] = { 2, 5, 3 };
MPI_Aint disp[3] = { 0, 5*sizeof(int), 5*sizeof(int)+10*sizeof(double) };
MPI_Datatype oldtypes[3], newtype;
oldtypes[0] = MPI_INT;
oldtypes[1] = MPI_DOUBLE;
oldtypes[2] = MPI_CHAR;

MPI_Type_struct( 3, blocklens, disp, oldtypes, &newtype );
MPI_Type_commit( &newtype );

MPI_Pack_size( 1, newtype, MPI_COMM_WORLD, &psize );
printf("pack size = %d\n", psize);
```

Exercise 5: 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).

What is not covered ……

- ❑ Communicator and topology
- ❑ Single-sided Communications
- ❑ Remote Memory Access
- ❑ Hybrid Programming: MPI + OpenMP, MPI + OpenACC, MPI + CUDA, ……
- ❑ MPI-based library
- ❑ MPI I/O
- ❑ MPI with other languages: python, perl, R, ……

Further information

❑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*

❑Help

help@scv.bu.edu

shaohao@bu.edu