MPI

Spring Training 2025

Yoshihiro Izawa

2025/05/13

Contents

Contents

1. MPI Overview

1.1 What is MPI (Message Passing Interface)?

- **A standard API** for message passing between distributed memories in parallel computing.
- MPI assumes a **distributed-memory computing system**
- MPI can run on **shared-memory computing system**
- MPI programming model (basically) uses **SIMD**

1.2 Parallel Programming Classification

- **Multi-Process**: MPI(Message Passing Interface), HPF(High Performance Fortran)
- **Multi-Thread**: OpenMP, Pthread(POSIX Thread)

- **Communication Model**:
  - Uses message passing for communication between processes.
- **Distributed Memory Support**:
  - Each process has its own memory space, no shared memory.
- **Multi-node Capacity**:
  - Can run across multiple nodes; abstracts network communication.
- **Standardized API**:
  - Standardized interface in C, C++, and Fortran; hightly portable.
- **Multiple Implementation**:
  - Available implementations include OpenMPI, MPICH, and Intel MPI, etc.
- **Difficalt to Debug**:
  - Debugging is challenging due to concurrency and communiaction complexity.

1.4 Typical example of Usage

- Simulation on a supercomputer(Physics, Meteorology, Chemistry, etc.)
- Data processing in large-scale data analysis (e.g., genomics, astronomy).
- Machine learning training on large datasets (e.g., distributed deep learning).

1.5 Comparison between implementations

	OpenMPI	MPICH	Intel MPI
Developer	Universities, Companies	Argonne National Laboratory	Intel Corporation
Distribution	Open source	Open source	Free version included
Optimization Target	General purpose	Lightweight, stable	Optimized for Intel architecture
Performance	Medium to high	Lightweight, stable, scalable	Best performance on Intel CPUs
Main Use	Academic clusters, general HPC	Research, education	Commercial HPC, Intel clusters

1.6 Key Communication Primitives

- **System function**: `MPI_Init`, `MPI_Finalize`, `MPI_Comm_size`, `MPI_Comm_rank`
- **Point-to-point communication**: `MPI_Send`, `MPI_Recv`
- **Collective communication**: `MPI_Bcast`, `MPI_Reduce`, `MPI_Alltoall`
- **Synchronization**: `MPI_Barrier`, `MPI_Wait`, `MPI_Test`
- **Derived data types**: `MPI_Type_create_struct`, `MPI_Type_vector`
- **Non-blocking communication**: `MPI_Isend`, `MPI_Irecv`
- **Remote memory access**: `MPI_Put`, `MPI_Get`
- **Process management**: `MPI_Comm_spawn`, `MPI_Comm_free`

2. Basic Learning of MPI

2.1.1 Hello World (C)

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

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    int num_procs;
    int my_rank;

    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    printf("Num of Proc : %d\n", num_procs);
    printf("My Rank     : %d\n", my_rank);

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

```
mpicc mpi_hello.c -o mpi_hello
mpirun -np 4 ./mpi_hello
Num of Proc : 4
My Rank     : 3
Num of Proc : 4
My Rank     : 2
Num of Proc : 4
My Rank     : 0
Num of Proc : 4
My Rank     : 1
```

2.1.2 Hello World (C++)

```cpp
#include <mpi.h>
#include <iostream>
#include <cstdlib>

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    int num_procs;
    int my_rank;

    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    std::cout << "Num of Proc : " << num_procs <<
std::endl;
    std::cout << "My Rank     : " << my_rank << std::endl;

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

```
mpic++ mpi_hello.cpp -o mpi_hello
mpirun -np 4 ./mpi_hello
Num of Proc : 4
My Rank     : 3
Num of Proc : 4
My Rank     : 1
Num of Proc : 4
My Rank     : 0
Num of Proc : 4
My Rank     : 2
```

2.1.3 Hello World (Fortran)

```
program hello_mpi
  use mpi
  implicit none

  integer :: ierr, rank, size

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

  print *, "Num of Proc:", size
  print *, "My Rank:    ", rank

  call MPI_Finalize(ierr)
end program hello_mpi
```

```
mpif90 mpi_hello.f90 -o mpi_hello
mpirun -np 4 ./mpi_hello
Num of Proc:  4
My Rank:      2
Num of Proc:  4
My Rank:      0
Num of Proc:  4
My Rank:      3
Num of Proc:  4
My Rank:      1
```

2.1.4 MPI Language Differences

	C	C++ (※)	Fortran
MPI Header	`#include <mpi.h>`	`#include <mpi.h>`	`use mpi` or `include 'mpif.h'`
Official MPI support	◯	▲	◯
Syntax intuitiveness	Explicit C syntax	Almost same as C	`call` and `subroutine` based
Compiler	`mpicc`	`mpicxx` or `mpic++`	`mpif90` or `mpifort`
Scientific computing	◯	▲	◎

- C++
  - MPI-3.0 abolished C++ only bindings.
  - Currently, C++ also uses C interface.
- Fortran
  - Considering readability, type safety, and portability, `use mpi` is recommended.

2.2.1 Overview

- **Process**:
  - computing unit in parallel computing in MPI.
  - process num is determined by `mpirun -np`
- **Group**:
  - a set of processes that can communicate with each other.
- **Communicator**:
  - a group of processes that can communicate with each other.
- **Rank**:
  - unique identifier for each process in MPI.
  - Ranks are assigned from 0 to `num_procs - 1`.

2.2.2 Communicator Image

- Each process belongs to some **group**.
- A group is associated with a **communicator**.
- Each process in a communicator has a unique **rank**.

```
- Eaxmple:
  - Process 0, 1, 2, 3 belong to a group.
  - Communicator: MPI_COMM_WORLD
  - Group: [P0, P1, P2, P3]
  - Rank: 0, 1, 2, 3

┌───────────────────────────────────────────┐
│       Communicator: MPI_COMM_WORLD        │
│       Group: [P0, P1, P2, P3]             │
│       Rank:  0    1    2    3             │
└───────────────────────────────────────────┘
```

2.2.3 MPI Functions for Communicator

- **MPI_COMM_WORLD**:
  - the default communicator that includes all processes.
  - all processes first belong to this communicator.
  - becomes the default communicator for most MPI functions.
- **MPI_Comm_rank**:
  - retrieves the rank of the calling process in specified communicator.
  - usually use `MPI_COMM_WORLD` as the communicator.
- **MPI_Comm_size**:
  - retrieves the number of processes in the specified communicator.
  - usually use `MPI_COMM_WORLD` as the communicator.
- **MPI_Comm_split**:
  - creates a new communicator by splitting the existing one based on a color and key.
  - in other words, create a new communicator with the same color processes.

- **MPI_Comm_rank** and **MPI_Comm_size**

```c
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("I am process %d\n", rank);
```

- **MPI_Comm_size**

```c
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("There are %d processes\n", size);
```

- **MPI_Comm_split**

```c
// ランクの3の剰余を基に color=0, 1, 2 に分ける
int color = rank % 3;

MPI_Comm new_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, rank, &new_comm);

int new_rank, new_size;
MPI_Comm_rank(new_comm, &new_rank);
MPI_Comm_size(new_comm, &new_size);

printf("World Rank %d => Group %d, New Rank %d of %d\n",
        rank, color, new_rank, new_size);

MPI_Comm_free(&new_comm);  // 新しいコミュニケータの解放
```

```
$ mpicc comm_split.c -o comm_split
$ mpirun -np 8 ./comm_split

World Rank 5 => Group 2, New Rank 1 of 2
World Rank 2 => Group 2, New Rank 0 of 2
World Rank 1 => Group 1, New Rank 0 of 3
World Rank 4 => Group 1, New Rank 1 of 3
World Rank 7 => Group 1, New Rank 2 of 3
World Rank 3 => Group 0, New Rank 1 of 3
World Rank 6 => Group 0, New Rank 2 of 3
World Rank 0 => Group 0, New Rank 0 of 3
```

World Rank	color(= rank % 3)	new group	new rank	new_size
0	0	{0, 3, 6}	0	3
1	1	{1, 4, 7}	0	3
2	2	{2, 5}	0	2
3	0	{0, 3, 6}	1	3
4	1	{1, 4, 7}	1	3
5	2	{2, 5}	1	2
6	0	{0, 3, 6}	2	3
7	1	{1, 4, 7}	2	3

2.3 Point-to-Point Communication

send_data()

```c
int send_data[10];
for (int i = 0; i < 10; i++)
    send_data[i] = i + 1;
int data_count = 10;

printf("Rank 0: Sending data.\n");
printf("send_data: [");
for (int i = 0; i < 10; i++)
    printf(" %d", send_data[i]);
printf(" ]\n");

MPI_Send((void*)send_data, data_count, MPI_INT,
1, 0, MPI_COMM_WORLD);
```

recv_data()

```c
int data[10];
int data_count = 10;
MPI_Status st;

printf("Rank 1: Receiving data.\n");
MPI_Recv((void*)data, data_count, MPI_INT, 0, 0,
MPI_COMM_WORLD, &st);
printf("recv_data: [");
for (int i = 0; i < 10; i++)
    printf(" %d", data[i]);
printf(" ]\n");
```

```
$ mpicc send_recv.c -o mpi_send_recv
$ mpirun -np 2 ./mpi_send_recv

Rank 0: Sending data.
send_data: [ 1 2 3 4 5 6 7 8 9 10 ]
Rank 1: Receiving data.
recv_data: [ 1 2 3 4 5 6 7 8 9 10 ]
```

- We can combine `send_data()` and `recv_data()` into a single program.
- The program can be run with `mpirun -np 2 ./mpi_send_recv2`

```
int data_count = 10;
int numbers[data_count];
if (world_rank == 0) {
    for (int i = 0; i < 10; i++) numbers[i] = i + 1;
    printf("Send Data:");
    for (int i = 0; i < 10; i++)
        printf(" %d", numbers[i]);
    printf("\n");
    MPI_Send((void *)&numbers, data_count, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&numbers, data_count, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Received Data:");
    for (int i = 0; i < 10; i++)
        printf(" %d", numbers[i]);
    printf("\n");
}
```

```
$ mpicc send_recv2.c -o mpi_send_recv2
$ mpirun -np 2 ./mpi_send_recv2

Send Data: 1 2 3 4 5 6 7 8 9 10
Received Data: 1 2 3 4 5 6 7 8 9 10
```

- many MPI functions have the following signature:

```c
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator
);
```

```c
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status
);
```

- Many MPI functions have the following signature

```
MPI_Send(
    void* data,                // data buffer address
    int count,                 // number of elements in the buffer
    MPI_Datatype datatype,     // data type of the elements
    int destination,           // destination process rank
    int tag,                   // message tag (for filtering)
    MPI_Comm communicator      // communicator
);
```

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

MPI Data Type	C Type
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	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_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

3. Collective Coommunication

3.1 syncronization

- Collective communication is a communication method that involves all processes in a communicator.
- In collective communiaction, syncronization among all process is required.
- All process cannot proceed until all processes reach the same point.
- To achieve this, MPI provides several collective communication functions.

3.2 MPI_Barrier

- **MPI_Barrier** is a collective communication function that synchronizes all processes in a communicator.
- All processes must call `MPI_Barrier` to ensure that all processes reach the same point before proceeding.
- It is often used to ensure that all processes have completed their previous tasks before moving on to the next step.
- The most basic usage of `MPI_Barrier` is to precise time measurement.
- If you do not call `MPI_Barrier` in all processed, the program will block and cannot proceed.
- `MPI_Barrier(MPI_Comm communicator);`

3.2 MPI_Barrier

```c
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

printf("Rank %d: before barrier\n", rank);

sleep(rank);

MPI_Barrier(MPI_COMM_WORLD);
printf("Rank %d: after barrier\n", rank);
```

```
$ mpicc barrier.c -o barrier
$ mpirun -np 4 ./barrier

Rank 2: before barrier
Rank 3: before barrier
Rank 0: before barrier
Rank 1: before barrier
Rank 3: after barrier
Rank 1: after barrier
Rank 0: after barrier
Rank 2: after barrier
```

3.3 MPI_Bcast

- **MPI_Bcast** is a collective communication function that broadcasts data from one process to all other processes in a communicator.
- It is used to distribute data from a root process to all other processes.
- All processes in the communicator must call `MPI_Bcast` with the same parameters.

```
MPI_Bcast(
    void* data,              // data buffer address
    int count,               // number of elements in the buffer
    MPI_Datatype datatype,   // data type of the elements
    int root,                // rank of the root process
    MPI_Comm communicator    // communicator
);
```

- Root Process: Sends the data to all other processes.
- Other Processes: Receive the data from the root process.

3.3 MPI_Bcast

```c
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int data[10];
if (rank == 0) {
    for (int i = 0; i < 10; i++) {
        data[i] = i + 1;
    }
    printf("Rank 0: broadcasting data = [");
    for (int i = 0; i < 10; i++)
        printf(" %d", data[i]);
    printf(" ]\n");
}

MPI_Bcast(data, 10, MPI_INT, 0, MPI_COMM_WORLD);

printf("Rank %d: received data = [", rank);
for (int i = 0; i < 10; i++)
    printf(" %d", data[i]);
printf(" ]\n");
```

```
$ mpicc bcast.c -o bcast
$ mpirun -np 4 ./bcast

Rank 0: broadcasting data = [ 1 2 3 4 5 6 7 8 9
10 ]
Rank 0: received data = [ 1 2 3 4 5 6 7 8 9 10 ]
Rank 2: received data = [ 1 2 3 4 5 6 7 8 9 10 ]
Rank 3: received data = [ 1 2 3 4 5 6 7 8 9 10 ]
Rank 1: received data = [ 1 2 3 4 5 6 7 8 9 10 ]
```

3.3 MPI_Bcast

- We can implement `MPI_Bcast` wrapper using `MPI_Send` and `MPI_Recv`.

```
void my_bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    for (int i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
      }
    }
  } else {
    MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
  }
}
```

- Q: Is this equivalent to `MPI_Bcast`?

- A: **No**, it is less efficient than `MPI_Bcast`.
- This implementation has only one network communication link. The process with `root` rank sends data to all other processes one by one.
- `MPI_Bcast` uses Tree-based broadcast algorithm.

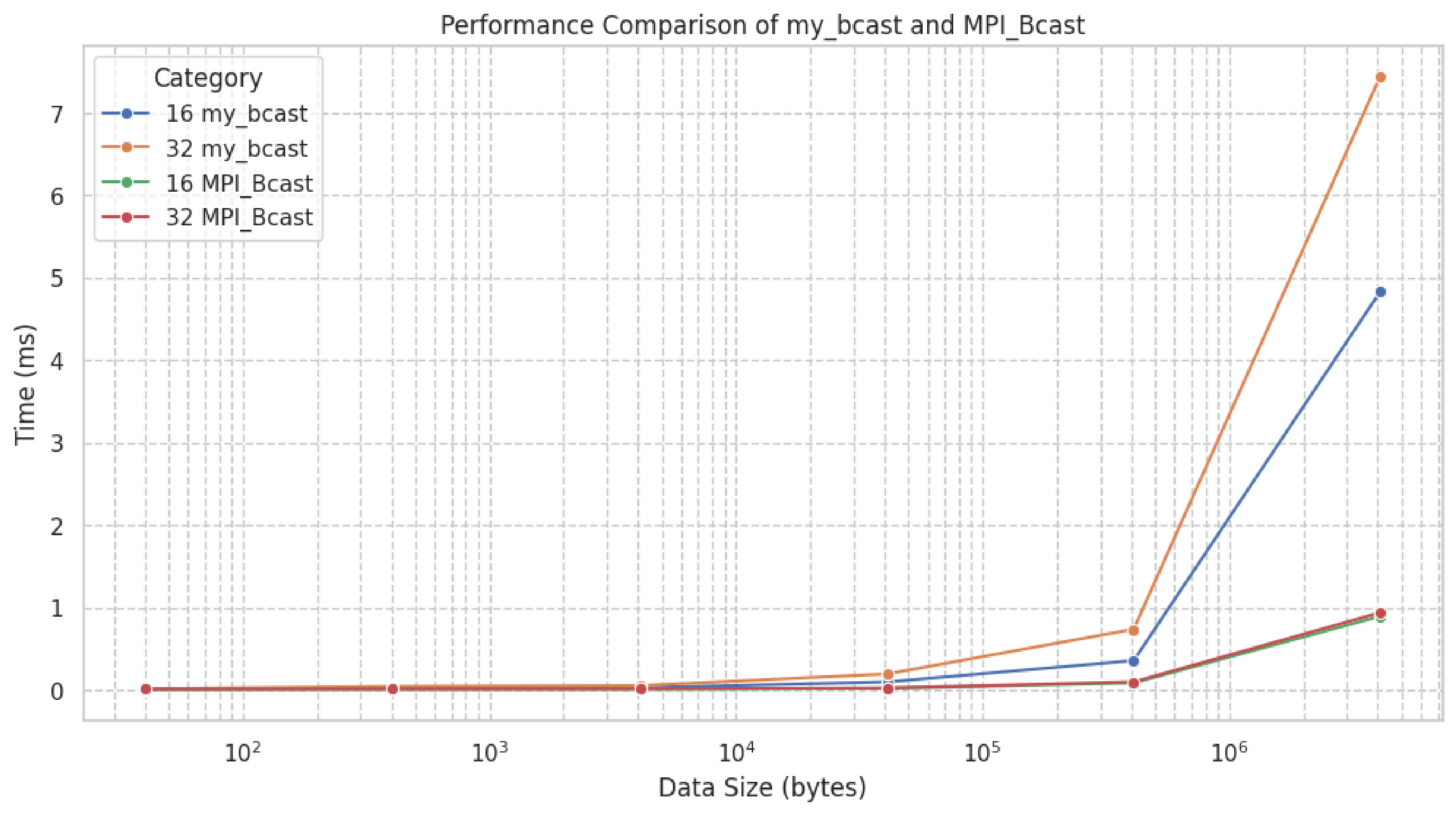
1. The root process sends data to process 1.
2. The root process sends data to process 2, process 1 sends data to process 3.
3. The root process sends data to process 4, process 1 sends data to process 5, process 2 sends data to process 6, process 3 sends data to process 7.

…

- Comparison of `MPI_Bcast` and `my_bcast`
- Average time of 10 trials

Procs	Data Size	my_bcast (ms)	MPI_Bcast (ms)
16	40	0.008	0.009
16	400	0.022	0.009
16	4k	0.026	0.010
16	40k	0.096	0.018
16	400k	0.355	0.084
16	4000k	4.832	0.893
32	40	0.012	0.012
32	400	0.041	0.011
32	4k	0.052	0.013
32	40k	0.193	0.022
32	400k	0.735	0.097
32	4000k	7.447	0.937

3.3 MPI_Bcast



Performance Comparison of my_bcast and MPI_Bcast
Category
16 my_bcast
32 my_bcast
16 MPI_Bcast
32 MPI_Bcast
Time (ms)
0
1
2
3
4
5
6
7
10^2
10^3
10^4
10^5
10^6
Data Size (bytes)

3.4.1 MPI_Scatter

- **MPI_Scatter** is a collective communication function that distributes data from a root process to all other processes in a communicator.
- `MPI_Bcast` sends the same data to all processes, while `MPI_Scatter` sends different chunks of data to each process.

```
MPI_Scatter(
    void* send_data,              // data buffer address to send
    int send_count,               // number of elements to send to each process
    MPI_Datatype send_datatype, // data type of the elements to send
    void* recv_data,              // data buffer address to receive
    int recv_count,               // number of elements to receive
    MPI_Datatype recv_datatype, // data type of the elements to receive
    int root,                     // rank of the root process
    MPI_Comm communicator.        // communicator
);
```

- `send_count`: the number of elements to send to each process.
- `recv_count`: the number of elements to receive from each process.

3.4 MPI_Scatter, MPI_Gather, MPI_Allgather

```c
#define TOTAL_DATA 20
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int send_data[TOTAL_DATA];
int recv_count = TOTAL_DATA / size;
int recv_data[recv_count];

if (rank == 0) {
    for (int i = 0; i < TOTAL_DATA; i++)
        send_data[i] = i;
    printf("Rank 0: Scattering data...\n");
}

MPI_Scatter(send_data, recv_count, MPI_INT,
            recv_data, recv_count, MPI_INT,
            0, MPI_COMM_WORLD);

printf("Rank %d received:", rank);
for (int i = 0; i < recv_count; i++)
    printf(" %d", recv_data[i]);
printf("\n");
```

```
// send_data == 5

$ mpicc scatter.c -o scatter
$ mpirun -np 4 ./scatter

Rank 0: Scattering data...
Rank 0 received: 0 1 2 3 4
Rank 1 received: 5 6 7 8 9
Rank 2 received: 10 11 12 13 14
Rank 3 received: 14 15 16 17 18
```

if `send_data` cannot divide by `size`, the last process will receive the remaining data.

```
// send_data == 6

Rank 0 received: 0 1 2 3 4 5
Rank 1 received: 6 7 8 9 10 11
Rank 2 received: 12 13 14 15 16 17
Rank 3 received: 18 19 20 -875497504 65535 20
```

3.4.2 MPI_Gather

- **MPI_Gather** is a collective communication function that collects data from all processes in a communicator and sends it to a root process.
- It is the reverse operation of `MPI_Scatter`.
- This is used in parallel sorting, parallel searching, and other parallel algorithms.

```
MPI_Gather(
    void* send_data,             // data buffer address to send
    int send_count,              // number of elements to send from each process
    MPI_Datatype send_datatype,  // data type of the elements to send
    void* recv_data,             // data buffer address to receive
    int recv_count,              // number of elements to receive from each process
    MPI_Datatype recv_datatype,  // data type of the elements to receive
    int root,                    // rank of the root process
    MPI_Comm communicator        // communicator
);
```

- Except for the root process, pass `NULL` for `recv_data` is allowed.
- `recv_count` is the number of elements to receive from each process, not the total number of elements.

3.4 MPI_Scatter, MPI_Gather, MPI_Allgather

```c
#define ITEMS_PER_PROC 2
...
int send_data[ITEMS_PER_PROC];
send_data[0] = rank * 2;
send_data[1] = rank * 2 + 1;

int recv_data[ITEMS_PER_PROC * size];

MPI_Gather(send_data, ITEMS_PER_PROC, MPI_INT,
            recv_data, ITEMS_PER_PROC, MPI_INT,
            0, MPI_COMM_WORLD);

if (rank == 0) {
    printf("Rank 0 gathered data: ");
    for (int i = 0; i < ITEMS_PER_PROC * size;
i++)
        printf("%d ", recv_data[i]);
    printf("\n");
} else {
    printf("Rank %d sent data: %d %d\n", rank,
send_data[0], send_data[1]);
}
```

```
$ mpicc gather.c -o gather
$ mpirun -np 4 ./gather

Rank 1 sent data: 2 3
Rank 3 sent data: 6 7
Rank 2 sent data: 4 5
Rank 0 gathered data: 0 1 2 3 4 5 6 7
```

3.4.3 MPI_Allgather

- `MPI_Scatter` and `MPI_Gather` conduct many-to-one or one-to-many communication.
- It is useful if you send data from multiple processed to multiple processes.
- **MPI_Allgather** is a collective communication function that collects data from all processes in a communicator and sends it to all other processes.
- It is like first `MPI_Gather` and then `MPI_Bcast`. Collect data by process rank order.

```
MPI_Allgather(
    void* send_data,               // data buffer address to send
    int send_count,                // number of elements to send from each process
    MPI_Datatype send_datatype, // data type of the elements to send
    void* recv_data,               // data buffer address to receive
    int recv_count,                // number of elements to receive from each process
    MPI_Datatype recv_datatype, // data type of the elements to receive
    MPI_Comm communicator          // communicator
);
```

- `MPI_Allgather` does not have a root process parameter.

```c
#define ITEMS_PER_PROC 2
...
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int send_data[ITEMS_PER_PROC];
send_data[0] = rank * 10;
send_data[1] = rank * 10 + 1;

int recv_data[ITEMS_PER_PROC * size];

MPI_Allgather(send_data, ITEMS_PER_PROC,
MPI_INT,
              recv_data, ITEMS_PER_PROC,
MPI_INT,
              MPI_COMM_WORLD);

printf("Rank %d received:", rank);
for (int i = 0; i < ITEMS_PER_PROC * size; i++)
    printf(" %d", recv_data[i]);
printf("\n");
```

```
$ mpicc allgather.c -o allgather
$ mpirun -np 4 ./allgather

Rank 2 received: 0 1 10 11 20 21 30 31
Rank 3 received: 0 1 10 11 20 21 30 31
Rank 0 received: 0 1 10 11 20 21 30 31
Rank 1 received: 0 1 10 11 20 21 30 31
```

3.5.1 MPI_Reduce

- `reduce` is a basic concept in functional programming. It transforms a set of numbers into a smaller set of numbers.
  - `reduce([1, 2, 3, 4, 5], sum) = 15`
  - `reduce([1, 2, 3, 4, 5], multiply) = 120`
- Collect distribulted data and apply a reduction operation is a tough task. However, MPI provides a simple interface to do this.
- **MPI_Reduce** is a collective communication function that collects data from all processes in a communicator to the root process, and applies a reduction operation to the data.

3.5 MPI_Reduce, MPI_Allreduce

```
MPI_Reduce(
    void* send_data,         // data buffer address to send
    void* recv_data,         // data buffer address to receive
    int count,               // number of elements to send from each process
    MPI_Datatype datatype,   // data type of the elements to send
    MPI_Op op,               // reduction operation to apply
    int root,                // rank of the root process
    MPI_Comm communicator    // communicator
);
```

- MPI Reduction Operations:
  - MPI_MAX - maximum value
  - MPI_MIN - minimum value
  - MPI_SUM - sum of values
  - MPI_PROD - product of values
  - MPI_LAND - logical AND of values
  - MPI_LOR - logical OR of values
  - MPI_BAND - bitwise AND of values
  - MPI_BOR - bitwise OR of values
  - MPI_MAXLOC - maximum value and its rank

3.5 MPI_Reduce, MPI_Allreduce

```c
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int value = rank + 1;
int sum;

MPI_Reduce(&value, &sum, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

double average = (double)sum / size;
if (rank == 0) {
    printf("Rank %d: sum = %d, avg = %.2f\n",
rank, sum, average);
} else {
    printf("Rank %d: sum = %d, avg = %.2f\n",
rank, sum, average);
}
```

```
$ mpicc reduce.c -o reduce
$ mpirun -np 4 ./reduce

Rank 1: sum = 4197236, avg = 1049309.00
Rank 0: sum = 10, avg = 2.50
Rank 2: sum = 4197236, avg = 1049309.00
Rank 3: sum = 4197236, avg = 1049309.00
```

3.5.2 MPI_Allreduce

- **MPI_Allreduce** is a collective communication function that collects data from all processes in a communicator, applies a reduction operation to the data, and distributes the result to all processes.
- It is similar to `MPI_Reduce`, but the result is available to all processes, not just the root process.
- This is useful when all processes need to know the result of the reduction operation.

```
MPI_Allreduce(
    void* send_data,          // data buffer address to send
    void* recv_data,          // data buffer address to receive
    int count,                // number of elements to send from each process
    MPI_Datatype datatype,    // data type of the elements to send
    MPI_Op op,                // reduction operation to apply
    MPI_Comm communicator     // communicator
);
```

3.5 MPI_Reduce, MPI_Allreduce

```c
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int value = rank + 1;
int total_sum = 0;

MPI_Allreduce(&value, &total_sum, 1, MPI_INT,
MPI_SUM, MPI_COMM_WORLD);

double average = (double)total_sum / size;
printf("Rank %d: total sum = %d, average =
%.2f\n", rank, total_sum, average);
```

```
$ mpicc allreduce.c -o allreduce
$ mpirun -np 4 ./allreduce

Rank 2: total sum = 10, average = 2.50
Rank 1: total sum = 10, average = 2.50
Rank 0: total sum = 10, average = 2.50
Rank 3: total sum = 10, average = 2.50
```

4. References

- MPI「超」入門（C 言語編）- 東京大学情報基盤センター
- 並列プログラミング入門
- MPI Tutorial