Day17

Table of Contents

1. Scripts

1.1. compile script

```
#!/bin/sh

#source /opt/ohpc/pub/apps/spack/share/spack/setup-env.sh
#spack load gcc/5i5y5cb
#spack load openmpi/c7kvqyq
```

```
source ~/git/spack/share/spack/setup-env.sh
spack load openmpi

inputFile=$1
outputFile="${1%.*}.out"      # extract the name of the file without extension and adding extension .out
#cmd=`mpicc $inputFile -o $outputFile`
cmd="mpicc $inputFile -o $outputFile"     # running code using MPI
echo "------------------------------------------------------------------"
echo "Command executed: $cmd"
echo "------------------------------------------------------------------"
$cmd

echo "Compilation successful. Check at $outputFile"
echo "------------------------------------------------------------------"
```

1.2. run script

```
#!/bin/sh

#source /opt/ohpc/pub/apps/spack/share/spack/setup-env.sh
#spack load gcc/5i5y5cbc
source ~/git/spack/share/spack/setup-env.sh
spack load openmpi

cmd="mpirun -np $2 $1"
echo "------------------------------------------------------------------"
echo "Command executed: $cmd"
echo "------------------------------------------------------------------"
echo "##################################################################"
echo "##########                    OUTPUT                    ##########"
echo "##################################################################"
echo
mpirun -np $2 $1
echo
echo "##################################################################"
echo "##########                     DONE                     ##########"
echo "##################################################################"
```

2. MPI_Type_struct

`MPI_Type_struct` (now deprecated and replaced by `MPI_Type_create_struct`) allows you to create a new MPI datatype that consists of a sequence of blocks, each with potentially different types and sizes. This is useful for sending or receiving complex data structures, such as structs in C.

2.1. Syntax (Deprecated)

```
int MPI_Type_struct(int count, const int array_of_blocklengths[], const MPI_Aint array_of_displacements[], const MPI_Datatype array_of_types[],
```

2.2. Syntax (Current)

```
int MPI_Type_create_struct(int count, const int array_of_blocklengths[], const MPI_Aint array_of_displacements[], const MPI_Datatype array_of_ty
```

- `count`: Number of blocks.
- `array_of_blocklengths`: Array specifying the number of elements in each block.
- `array_of_displacements`: Array specifying the byte displacement of each block from the start.
- `array_of_types`: Array specifying the datatype of each block.
- `newtype`: New datatype representing the struct.

2.3. Example Code

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

typedef struct {
    int a;
    double b;
    char c;
} my_struct;

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

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

    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for this example\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    my_struct data;
    MPI_Datatype struct_type;

    // Create the datatype for my_struct
    int blocklengths[3] = {1, 1, 1};
    MPI_Aint displacements[3];
    MPI_Datatype types[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
```

```
    displacements[0] = offsetof(my_struct, a);
    displacements[1] = offsetof(my_struct, b);
    displacements[2] = offsetof(my_struct, c);

    MPI_Type_create_struct(3, blocklengths, displacements, types, &struct_type);
    MPI_Type_commit(&struct_type);

    if (rank == 0) {
        data.a = 42;
        data.b = 3.14;
        data.c = 'A';

        MPI_Send(&data, 1, struct_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent struct: {a = %d, b = %.2f, c = %c}\n", data.a, data.b, data.c);
    } else if (rank == 1) {
        MPI_Recv(&data, 1, struct_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received struct: {a = %d, b = %.2f, c = %c}\n", data.a, data.b, data.c);
    }

    MPI_Type_free(&struct_type);
    MPI_Finalize();
    return 0;
}
```

2.4. Explanation

- **Initialization**: Initialize MPI, get the rank and size of the communicator.
- **Datatype Creation**:
  - `blocklengths` specifies the number of elements in each block: {1, 1, 1}.
  - `displacements` specifies the byte offsets of each block within the struct: calculated using `offsetof`.
  - `types` specifies the datatype of each block: {MPI_INT, MPI_DOUBLE, MPI_CHAR}.
  - `MPI_Type_create_struct` creates a new datatype `struct_type` representing the `my_struct`.
- **Process 0**:
  - Initializes the `data` struct with values.
  - Sends the `data` struct using the `struct_type` to process 1.
  - Prints the `data` struct.
- **Process 1**:
  - Receives the struct from process 0 into the `data` struct using the `struct_type`.
  - Prints the `data` struct after receiving.
- **Datatype Cleanup**: Free the `struct_type` with `MPI_Type_free`.
- **Finalize**: Finalize the MPI environment.

2.5. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_type_struct.c
```

```
---------------------------------------------------------------
Command executed: mpicc mpi_type_struct.c -o mpi_type_struct.out
---------------------------------------------------------------
Compilation successful. Check at mpi_type_struct.out
---------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_type_struct.out 2
```

```
---------------------------------------------------------------
Command executed: mpirun -np 2 ./mpi_type_struct.out
---------------------------------------------------------------
###############################################################
##########                    OUTPUT                   ##########
###############################################################

Process 0 sent struct: {a = 42, b = 3.14, c = A}
Process 1 received struct: {a = 42, b = 3.14, c = A}

###############################################################
##########                    DONE                     ##########
###############################################################
```

This example demonstrates how to use `MPI_Type_create_struct` to communicate complex data structures in MPI.

3. MPI Type Struct with different blocklength

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

typedef struct {
    int arr[3];
    double b;
    char c;
} my_struct;
```

```c
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

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

    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for this example\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    my_struct data;
    MPI_Datatype struct_type;

    // Create the datatype for my_struct
    int blocklengths[3] = {2, 1, 1};  // Sending part of the array, the double, and the char
    MPI_Aint displacements[3];
    MPI_Datatype types[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};

    displacements[0] = offsetof(my_struct, arr);
    displacements[1] = offsetof(my_struct, b);
    displacements[2] = offsetof(my_struct, c);

    MPI_Type_create_struct(3, blocklengths, displacements, types, &struct_type);
    MPI_Type_commit(&struct_type);

    if (rank == 0) {
        data.arr[0] = 1;
        data.arr[1] = 2;
        data.arr[2] = 3;
        data.b = 3.14;
        data.c = 'A';

        MPI_Send(&data, 1, struct_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 has struct: {arr = [%d, %d, %d], b = %.2f, c = %c}\n", data.arr[0], data.arr[1], data.arr[2], data.b, data.c);
    } else if (rank == 1) {
        // Initialize the struct to zero
        data.arr[0] = data.arr[1] = data.arr[2] = 0;
        data.b = 0.0;
        data.c = '0';

        MPI_Recv(&data, 1, struct_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received struct: {arr = [%d, %d, %d], b = %.2f, c = %c}\n", data.arr[0], data.arr[1], data.arr[2], data.b, data.c);
    }

    MPI_Type_free(&struct_type);
    MPI_Finalize();
    return 0;
}
```

3.1. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_type_struct2.c
```

```
---------------------------------------------------------------
Command executed: mpicc mpi_type_struct2.c -o mpi_type_struct2.out
---------------------------------------------------------------
Compilation successful. Check at mpi_type_struct2.out
---------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_type_struct2.out 2
```

```
---------------------------------------------------------------
Command executed: mpirun -np 2 ./mpi_type_struct2.out
---------------------------------------------------------------
###############################################################
##########                    OUTPUT                   ##########
###############################################################

Process 0 has struct: {arr = [1, 2, 3], b = 3.14, c = A}
Process 1 received struct: {arr = [1, 2, 0], b = 3.14, c = A}

###############################################################
##########                    DONE                     ##########
###############################################################
```

This example demonstrates how to use `MPI_Type_create_struct` to communicate complex data structures in MPI, specifically how to send parts of an array along with other fields.

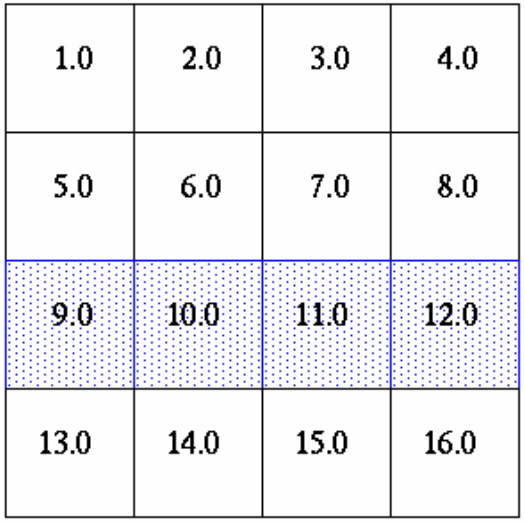
4. Reference

These images are for your reference. Link for your reference if you want to learn more about it.

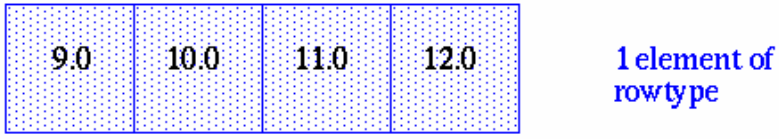
4.1. MPI Type contigous

MPI_ Type_contiguous

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```



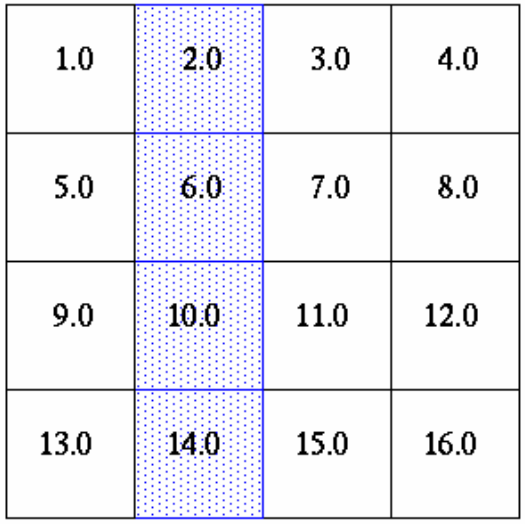

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```



4.2. MPI Type vector

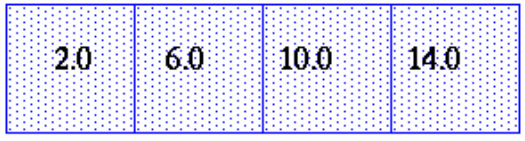
MPI_ Type_vector

count = 4; blocklength = 1; stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,
&columntype);



a[4][4]

MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);



1 element of
columntype

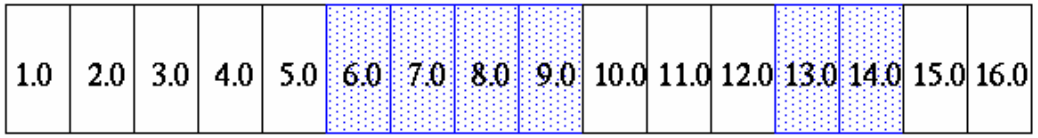
4.3. MPI Type indexed

MPI_ Type_indexed

count = 2;

blocklengths[0] = 4;
displacements[0] = 5;

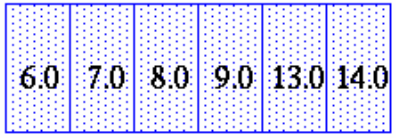
blocklengths[1] = 2;
displacements[1] = 12;



a[16]

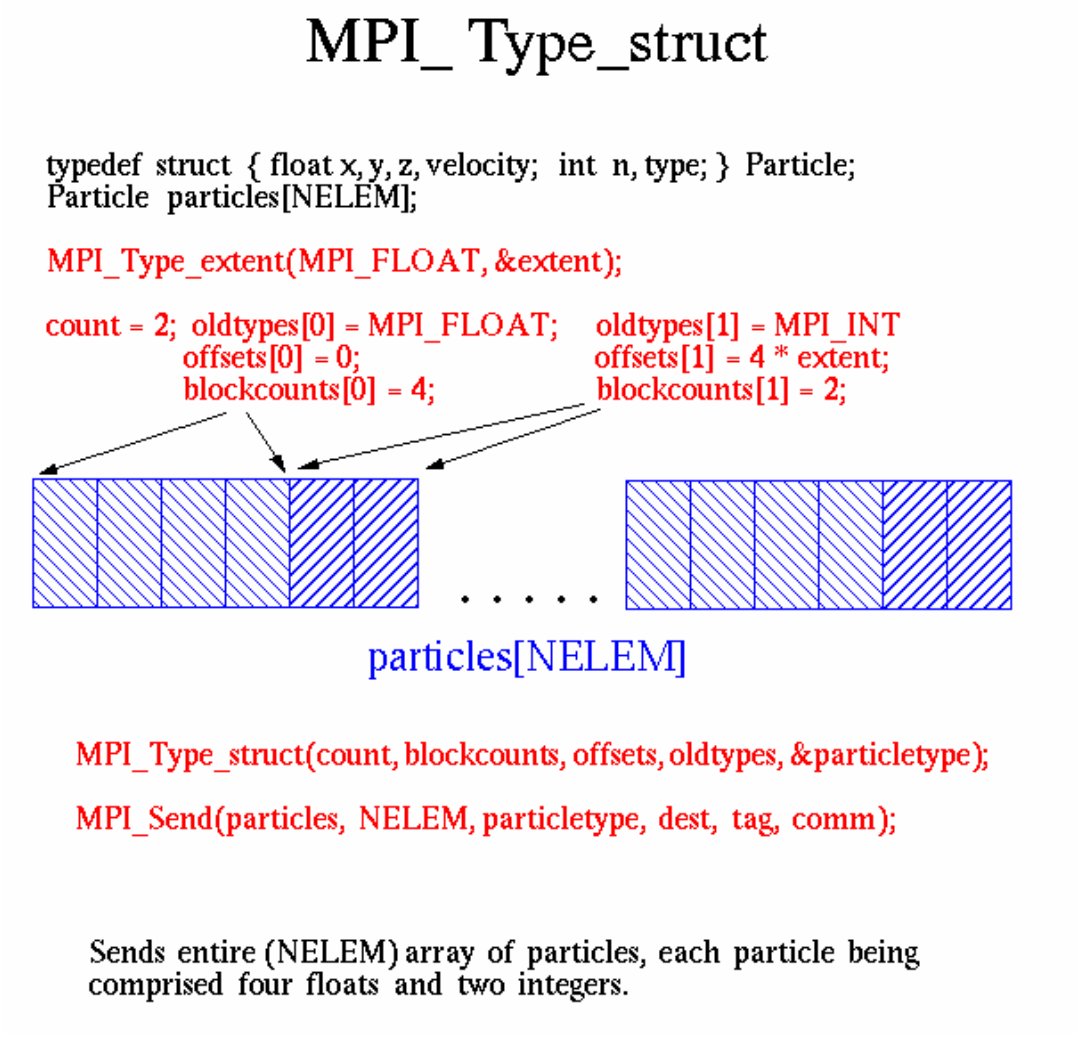
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);

MPI_Send(&a, 1, indextype, dest, tag, comm);



1 element of indextype

4.4. MPI Type struct



5. task1

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

typedef struct {
    char name[20];
    int prn;
    int age;
    char email[40];
    double salary;
    int marks[5];
} student;

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

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

    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for this example\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    student st;
    MPI_Datatype struct_type;

    // Create the datatype for my_struct
    int blocklengths[6] = {20, 1, 1, 40, 1, 5};
    MPI_Aint displacements[6];
    MPI_Datatype types[6] = {MPI_CHAR, MPI_INT, MPI_INT, MPI_CHAR, MPI_DOUBLE, MPI_INT};

    displacements[0] = offsetof(student, name);
    displacements[1] = offsetof(student, prn);
    displacements[2] = offsetof(student, age);
    displacements[3] = offsetof(student, email);
    displacements[4] = offsetof(student, salary);
    displacements[5] = offsetof(student, marks);

    MPI_Type_create_struct(6, blocklengths, displacements, types, &struct_type);
    MPI_Type_commit(&struct_type);

    if (rank == 0) {
        strcpy(st.name,"Abhishek Raj");
        st.prn = 001;
        st.age = 24;
        strcpy(st.email, "abhi@abhi.com");
        st.salary = 10000.011111;
        st.marks[0] = 40;
        st.marks[1] = 39;
```

```
        st.marks[2] = 40;
        st.marks[3] = 40;
        st.marks[4] = 38;

        MPI_Send(&st, 1, struct_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 send data: \n");
        printf("Name : %s\n", st.name);
        printf("PRN : %d\n", st.prn);
        printf("Age : %d\n", st.age);
        printf("Email : %s\n", st.email);
        printf("Salary : %lf\n", st.salary);
        printf("Marks : ");
        for(int i = 0; i < 5; i++) printf("%d ", st.marks[i]);
        printf("\n");
    } else if (rank == 1) {
        MPI_Recv(&st, 1, struct_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: \n");
        printf("Name : %s\n", st.name);
        printf("PRN : %d\n", st.prn);
        printf("Age : %d\n", st.age);
        printf("Email : %s\n", st.email);
        printf("Salary : %lf\n", st.salary);
        printf("Marks : ");
        for(int i = 0; i < 5; i++) printf("%d ", st.marks[i]);
        printf("\n");
    }

    MPI_Type_free(&struct_type);
    MPI_Finalize();
    return 0;
}
```

5.1. Compilation and Execution

- Compile the program:

```
bash compile.sh task1.c
```

```
-----------------------------------------------------------------
Command executed: mpicc task1.c -o task1.out
-----------------------------------------------------------------
Compilation successful. Check at task1.out
-----------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./task1.out 2
```

```
--------------------------------------------------------------------
Command executed: mpirun -np 2 ./task1.out
--------------------------------------------------------------------
######################################################################
##########                    OUTPUT                    ##########
######################################################################

Process 0 send data:
Name : Abhishek Raj
PRN : 1
Age : 24
Email : abhi@abhi.com
Salary : 10000.011111
Marks : 40 39 40 40 38
Process 1 received data:
Name : Abhishek Raj
PRN : 1
Age : 24
Email : abhi@abhi.com
Salary : 10000.011111
Marks : 40 39 40 40 38

######################################################################
##########                    DONE                      ##########
######################################################################
```

6. MPI Packing and Unpacking

MPI provides mechanisms for packing and unpacking data into a contiguous buffer. This is useful for sending complex data structures without creating a custom MPI datatype. Instead, you manually pack the data into a buffer and then send the buffer.

6.1. Functions

- `MPI_Pack`: Packs data of different types into a contiguous buffer.
- `MPI_Unpack`: Unpacks data from a contiguous buffer.

6.2. Syntax

```
int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm);
```

- `inbuf`: Input buffer containing data to be packed.
- `incount`: Number of elements in the input buffer.
- `datatype`: Datatype of each element in the input buffer.
- `outbuf`: Output buffer to contain packed data.
- `outsize`: Size of the output buffer.
- `position`: Current position in the output buffer (updated by MPI).
- `comm`: Communicator.

```
int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm);
```

6.3. Example Code

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

typedef struct {
    int a;
    double b;
    char c;
} my_struct;

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

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

    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for this example\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    my_struct data;
    int buffer_size, position;
    void *buffer;

    if (rank == 0) {
        data.a = 42;
        data.b = 3.14;
        data.c = 'A';

        // Calculate the buffer size required for packing
        MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &buffer_size);
        buffer_size += sizeof(double) + sizeof(char); // Adding the sizes of the other data types
        buffer = malloc(buffer_size);
```

```
        position = 0;
        MPI_Pack(&data.a, 1, MPI_INT, buffer, buffer_size, &position, MPI_COMM_WORLD);
        MPI_Pack(&data.b, 1, MPI_DOUBLE, buffer, buffer_size, &position, MPI_COMM_WORLD);
        MPI_Pack(&data.c, 1, MPI_CHAR, buffer, buffer_size, &position, MPI_COMM_WORLD);

        MPI_Send(buffer, buffer_size, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent packed data\n");

        free(buffer);
    } else if (rank == 1) {
        MPI_Status status;
        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_PACKED, &buffer_size);

        buffer = malloc(buffer_size);
        MPI_Recv(buffer, buffer_size, MPI_PACKED, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        position = 0;
        MPI_Unpack(buffer, buffer_size, &position, &data.a, 1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, buffer_size, &position, &data.b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
        MPI_Unpack(buffer, buffer_size, &position, &data.c, 1, MPI_CHAR, MPI_COMM_WORLD);

        printf("Process 1 received unpacked data: {a = %d, b = %.2f, c = %c}\n", data.a, data.b, data.c);
        free(buffer);
    }

    MPI_Finalize();
    return 0;
}
```

6.4. Explanation

- **Initialization**: Initialize MPI, get the rank and size of the communicator.
- **Process 0**:
    - Initializes the `data` struct with values.
    - Calculates the buffer size required for packing the data using `MPI_Pack_size` and manually adds the sizes of the other data types.
    - Allocates memory for the buffer.
    - Packs each member of the struct into the buffer using `MPI_Pack`.
    - Sends the packed buffer to process 1 using `MPI_Send`.
    - Frees the buffer memory.
- **Process 1**:
    - Uses `MPI_Probe` to get the size of the incoming message.
    - Allocates memory for the buffer based on the received size.
    - Receives the packed buffer from process 0 using `MPI_Recv`.
    - Unpacks each member of the struct from the buffer using `MPI_Unpack`.

    - Prints the unpacked data.
    - Frees the buffer memory.
- **Finalize**: Finalize the MPI environment.

6.5. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_pack_unpack.c
```

```
----------------------------------------------------------------
Command executed: mpicc mpi_pack_unpack.c -o mpi_pack_unpack.out
----------------------------------------------------------------
Compilation successful. Check at mpi_pack_unpack.out
----------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_pack_unpack.out 2
```

```
----------------------------------------------------------------
Command executed: mpirun -np 2 ./mpi_pack_unpack.out
----------------------------------------------------------------
################################################################
##########                    OUTPUT                    ##########
################################################################

Process 0 sent packed data
Process 1 received unpacked data: {a = 42, b = 3.14, c = A}

################################################################
##########                     DONE                     ##########
################################################################
```

This example demonstrates how to use `MPI_Pack` and `MPI_Unpack` to communicate complex data structures in MPI.

7. MPI Pack Size, Probe, and Get Count

7.1. MPI_Pack_size

MPI_Pack_size is used to calculate the size of the buffer needed to pack a message. This function helps ensure that the buffer you allocate is large enough to hold the packed data.

7.1.1. Syntax

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

- `incount`: Number of elements in the input buffer.
- `datatype`: Datatype of each element in the input buffer.
- `comm`: Communicator.
- `size`: Pointer to the size of the packed message (output parameter).

7.1.2. Example

Let’s calculate the buffer size for packing an integer, a double, and a char.

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

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

    int size_int, size_double, size_char, total_size;
    MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &size_int);
    MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD, &size_double);
    MPI_Pack_size(1, MPI_CHAR, MPI_COMM_WORLD, &size_char);

    total_size = size_int + size_double + size_char;
    printf("Buffer size required for packing: %d bytes\n", total_size);

    MPI_Finalize();
    return 0;
}
```

- Compile the program:

```
  bash compile.sh mpi_pack_size.c
```

```
----------------------------------------------------------------------
Command executed: mpicc mpi_pack_size.c -o mpi_pack_size.out
----------------------------------------------------------------------
```

```
Compilation successful. Check at mpi_pack_size.out
------------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_pack_size.out 2
```

```
------------------------------------------------------------------
Command executed: mpirun -np 2 ./mpi_pack_size.out
------------------------------------------------------------------
######################################################################
##########                    OUTPUT                    ##########
######################################################################

Buffer size required for packing: 13 bytes
Buffer size required for packing: 13 bytes

######################################################################
##########                     DONE                     ##########
######################################################################
```

7.2. MPI_Probe

MPI_Probe allows you to probe for an incoming message without actually receiving it. This can be useful to determine the size of the message and allocate an appropriately sized buffer.

7.2.1. Syntax

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- `source`: Rank of the source process (or `MPI_ANY_SOURCE` for any source).
- `tag`: Message tag (or `MPI_ANY_TAG` for any tag).
- `comm`: Communicator.
- `status`: Status object that contains information about the message (output parameter).

7.3. MPI_Get_count

MPI_Get_count retrieves the number of elements of a specific datatype in a message. This function is often used after probing to determine the exact size of the received message.

7.3.1. Syntax

```
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count);
```

- `status`: Status object returned by `MPI_Probe` or `MPI_Recv`.
- `datatype`: Datatype of each element in the message.
- `count`: Pointer to the number of received elements (output parameter).

7.4. Example

Let's combine `MPI_Probe` and `MPI_Get_count` to dynamically allocate a buffer for receiving a message.

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for this example\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    if (rank == 0) {
        int data[5] = {1, 2, 3, 4, 5};
        MPI_Send(data, 5, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data to process 1\n");
    } else if (rank == 1) {
        MPI_Status status;
        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
        int count;
        MPI_Get_count(&status, MPI_INT, &count);
        int *buffer = (int*)malloc(count * sizeof(int));
        MPI_Recv(buffer, count, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received %d integers:\n", count);
        for (int i = 0; i < count; i++) {
            printf("%d ", buffer[i]);
        }
        printf("\n");
        free(buffer);
    }
    MPI_Finalize();
    return 0;
}
```

7.5. Explanation

- **MPI_Pack_size**:
  - This function is called three times to calculate the size required for packing an integer, a double, and a char.
  - The sizes are then summed to determine the total buffer size needed for packing.
- **MPI_Probe**:
  - Process 1 uses `MPI_Probe` to check for an incoming message from process 0 without actually receiving it.
  - The `status` object is filled with information about the message.
- **MPI_Get_count**:
  - `MPI_Get_count` is called to determine the number of integers in the received message using the `status` object from `MPI_Probe`.
  - This allows process 1 to dynamically allocate a buffer of the appropriate size.

7.6. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_probe_get_count.c
```

```
----------------------------------------------------------------
Command executed: mpicc mpi_probe_get_count.c -o mpi_probe_get_count.out
----------------------------------------------------------------
Compilation successful. Check at mpi_probe_get_count.out
----------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_probe_get_count.out 2
```

```
----------------------------------------------------------------
Command executed: mpirun -np 2 ./mpi_probe_get_count.out
----------------------------------------------------------------
################################################################
##########                   OUTPUT                   ##########
################################################################

Process 0 sent data to process 1
Process 1 received 5 integers:
```

```
1 2 3 4 5

######################################################################
##########                     DONE                     ##########
######################################################################
```

8. Task

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

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

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

    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for this example\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if (rank != 0) {
        MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    } else {
        int data = 0;
        MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("I have received message from process %d\n", data);
    }

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh mpi_any_src_and_tag.c
```

```
-----------------------------------------------------------------
Command executed: mpicc mpi_any_src_and_tag.c -o mpi_any_src_and_tag.out
-----------------------------------------------------------------
Compilation successful. Check at mpi_any_src_and_tag.out
-----------------------------------------------------------------
```

```
bash run.sh ./mpi_any_src_and_tag.out 10
```

```
--------------------------------------------------------------------
Command executed: mpirun -np 10 ./mpi_any_src_and_tag.out
--------------------------------------------------------------------
######################################################################
##########                    OUTPUT                    ##########
######################################################################

I have received message from process 1

######################################################################
##########                     DONE                     ##########
######################################################################
```

9. Task v2

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

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

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

    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for this example\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if (rank != 0) {
        int data = 100;
        MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    } else {
        MPI_Status status;
        int data = 0;
        MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        printf("I have received message from process %d\n", status.MPI_SOURCE);
    }

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh mpi_any_src_and_tag.c
```

```
----------------------------------------------------------------
Command executed: mpicc mpi_any_src_and_tag.c -o mpi_any_src_and_tag.out
----------------------------------------------------------------
Compilation successful. Check at mpi_any_src_and_tag.out
----------------------------------------------------------------
```

```
bash run.sh ./mpi_any_src_and_tag.out 10
```

```
----------------------------------------------------------------
Command executed: mpirun -np 10 ./mpi_any_src_and_tag.out
----------------------------------------------------------------
################################################################
##########                   OUTPUT                   ##########
```

```
############################################################

I have received message from process 9

############################################################
##########                    DONE                    ##########
############################################################
```

10. MPI Groups and Communicators

In MPI, communicators and groups are essential for defining communication contexts and organizing processes. A communicator encapsulates a group of processes that can communicate with each other. Each process within a communicator has a unique rank, starting from 0.

10.1. Groups

A group is an ordered set of processes. Groups are used to define the members of a communicator. Groups are created from existing communicators and can be manipulated using various MPI functions.

10.2. Communicators

A communicator is a communication domain, and it is the primary context for MPI communication operations. The default communicator, `MPI_COMM_WORLD`, includes all the processes in an MPI program. You can create new communicators with different groups of processes.

10.3. Creating and Managing Groups and Communicators

- **Creating Groups**: You can create a new group from an existing communicator using `MPI_Comm_group`, which extracts the group from a communicator.
- **Creating Communicators**: You can create new communicators from existing groups using `MPI_Comm_create` or `MPI_Comm_split`.

10.4. Difference between Groups and Communicators

- **Groups**:
  - A group is a collection of processes identified by their ranks.
  - Groups do not have communication contexts.
  - Groups are used to create new communicators.
- **Communicators**:

    - A communicator includes a group and a communication context.
    - Communicators are used for performing communication operations.
    - The default communicator, `MPI_COMM_WORLD`, includes all processes.

10.5. MPI Syntax and Functions

10.5.1. MPI_Comm_split

This function splits an existing communicator into multiple, non-overlapping communicators based on the color and key values provided.

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
```

- **comm**: The original communicator.
- **color**: Determines the group to which a process belongs.
- **key**: Determines the rank within the new communicator.
- **newcomm**: The new communicator.

10.5.2. MPI_Comm_group

This function retrieves the group associated with a communicator.

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group);
```

- **comm**: The communicator.
- **group**: The group associated with the communicator.

10.5.3. MPI_Group_incl

This function creates a new group from a subset of processes in an existing group.

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup);
```

- **group**: The original group.
- **n**: Number of ranks in the new group.
- **ranks**: Array of ranks in the original group to include in the new group.
- **newgroup**: The new group.

10.5.4. MPI_Comm_create_group

This function creates a new communicator from a group.

```
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm);
```

- **comm**: The original communicator.
- **group**: The group defining the new communicator.
- **tag**: Tag for the new communicator.
- **newcomm**: The new communicator.

10.5.5. MPI_Group_free

This function deallocates a group.

```
int MPI_Group_free(MPI_Group *group);
```

- **group**: The group to be deallocated.

10.5.6. MPI_Comm_free

This function deallocates a communicator.

```
int MPI_Comm_free(MPI_Comm *comm);
```

- **comm**: The communicator to be deallocated.

10.6. Example: Creating and Using Groups and Communicators

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Split the world group into two groups
```

```
    int color = rank % 2;  // Determine color based on rank
    MPI_Comm new_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &new_comm);
    // Get the new rank and size in the new communicator
    int new_rank, new_size;
    MPI_Comm_rank(new_comm, &new_rank);
    MPI_Comm_size(new_comm, &new_size);
    printf("World Rank: %d, New Rank: %d, New Size: %d\n", rank, new_rank, new_size);
    // Perform some communication within the new communicator
    int send_data = new_rank;
    int recv_data;
    MPI_Allreduce(&send_data, &recv_data, 1, MPI_INT, MPI_SUM, new_comm);
    printf("World Rank: %d, New Comm Sum: %d\n", rank, recv_data);
    // Free the new communicator and group
    MPI_Comm_free(&new_comm);
    MPI_Finalize();
    return 0;
}
```

10.7. Explanation

1. **Extract World Group**:
   - `MPI_Comm_group` is used to get the group of `MPI_COMM_WORLD`.
2. **Split the Communicator**:
   - `MPI_Comm_split` is used to split `MPI_COMM_WORLD` into two new communicators based on the color value (rank modulo 2).
   - This creates two new communicators: one for even ranks and one for odd ranks.
3. **New Rank and Size**:
   - The new rank and size within the new communicator are obtained using `MPI_Comm_rank` and `MPI_Comm_size`.
4. **Communication**:
   - `MPI_Allreduce` is performed within the new communicator to compute the sum of ranks in the new communicator.
5. **Cleanup**:
   - The new communicator and group are freed using `MPI_Comm_free` and `MPI_Group_free`.

10.8. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_groups_communicators.c
```

```
------------------------------------------------------------------
Command executed: mpicc mpi_groups_communicators.c -o mpi_groups_communicators.out
------------------------------------------------------------------
Compilation successful. Check at mpi_groups_communicators.out
------------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_groups_communicators.out 5
```

```
------------------------------------------------------------------
Command executed: mpirun -np 5 ./mpi_groups_communicators.out
------------------------------------------------------------------
######################################################################
##########                    OUTPUT                    ##########
######################################################################

World Rank: 1, New Rank: 0, New Size: 2
World Rank: 0, New Rank: 0, New Size: 3
World Rank: 0, New Comm Sum: 3
World Rank: 4, New Rank: 2, New Size: 3
World Rank: 4, New Comm Sum: 3
World Rank: 2, New Rank: 1, New Size: 3
World Rank: 2, New Comm Sum: 3
World Rank: 3, New Rank: 1, New Size: 2
World Rank: 3, New Comm Sum: 1
World Rank: 1, New Comm Sum: 1

######################################################################
##########                     DONE                     ##########
######################################################################
```

This example demonstrates how to create and use groups and communicators in MPI to organize and manage process communication in parallel applications.

11. Task Parallelism

When working with MPI, it’s often necessary to divide tasks among different groups of processes. MPI provides various functions to create and manage groups and communicators.

11.1. Example: Task Parallelism with Groups and Communicators

This example demonstrates the use of the above functions to create two groups, assign communicators, and

perform different tasks.

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

void perform_computation(int rank) {
    printf("Process %d performing computation\n", rank);
    // Simulate computation by sleeping for a while
    sleep(2);
}

void perform_io_operations(int rank) {
    printf("Process %d performing I/O operations\n", rank);
    // Simulate I/O by sleeping for a while
    sleep(3);
}

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

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

    // Define two groups: one for even ranks and one for odd ranks
    int half_size = size / 2;
    int *even_ranks = malloc(half_size * sizeof(int));
    int *odd_ranks = malloc((size - half_size) * sizeof(int));

    int even_count = 0, odd_count = 0;
    for (int i = 0; i < size; i++) {
        if (i % 2 == 0) {
            even_ranks[even_count] = i;
            even_count++;
        } else {
            odd_ranks[odd_count++] = i;
        }
    }

    // Create groups
    MPI_Group world_group, even_group, odd_group;
    MPI_Comm_group(MPI_COMM_WORLD, &world_group);
    MPI_Group_incl(world_group, even_count, even_ranks, &even_group);
    MPI_Group_incl(world_group, odd_count, odd_ranks, &odd_group);

    // Create new communicators
    MPI_Comm even_comm, odd_comm;
    MPI_Comm_create_group(MPI_COMM_WORLD, even_group, 0, &even_comm);
    MPI_Comm_create_group(MPI_COMM_WORLD, odd_group, 1, &odd_comm);
```

```
    // Perform tasks based on the group
    if (rank % 2 == 0 && even_comm != MPI_COMM_NULL) {
        perform_computation(rank);
    } else if (rank % 2 != 0 && odd_comm != MPI_COMM_NULL) {
        perform_io_operations(rank);
    }

    // Free the groups and communicators
    MPI_Group_free(&even_group);
    MPI_Group_free(&odd_group);
    if (even_comm != MPI_COMM_NULL) MPI_Comm_free(&even_comm);
    if (odd_comm != MPI_COMM_NULL) MPI_Comm_free(&odd_comm);
    MPI_Group_free(&world_group);

    free(even_ranks);
    free(odd_ranks);

    MPI_Finalize();
    return 0;
}
```

11.2. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_task_parallelism_manual_groups.c
```

```
---------------------------------------------------------------------
Command executed: mpicc mpi_task_parallelism_manual_groups.c -o mpi_task_parallelism_manual_groups.out
---------------------------------------------------------------------
Compilation successful. Check at mpi_task_parallelism_manual_groups.out
---------------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_task_parallelism_manual_groups.out 9
```

```
---------------------------------------------------------------------
Command executed: mpirun -np 9 ./mpi_task_parallelism_manual_groups.out
---------------------------------------------------------------------
#####################################################################
##########                    OUTPUT                     ##########
#####################################################################
```

```
Process 0 performing computation
Process 8 performing computation
Process 1 performing I/O operations
Process 6 performing computation
Process 4 performing computation
Process 2 performing computation
Process 5 performing I/O operations
Process 7 performing I/O operations
Process 3 performing I/O operations

##################################################################
##########                    DONE                    ##########
##################################################################
```

11.3. Questions and Answers

11.3.1. Is there any way for two different groups to communicate with each other?

Yes, two different groups can communicate using an inter-communicator. `MPI_Intercomm_create` can be used to establish communication between two groups, allowing them to exchange messages.

11.3.2. What are the communication mechanisms in different groups and the same group?

- **Different Groups**:
  - **Inter-communicator**: Allows communication between different groups.
  - **Point-to-Point Communication**: Direct communication between processes in different groups using inter-communicators.
- **Same Group**:
  - **Intra-communicator**: Default communication within the same group using collective operations like `MPI_Bcast`, `MPI_Reduce`, etc.

11.3.3. Do the two groups have the same communicator?

No, each group will have its unique communicator. When groups are created from a world communicator, they each get a new communicator that allows them to operate independently. An inter-communicator is required for communication between these separate groups. Task parallelism focuses on distributing tasks (rather than data) across different processes. Each task can perform different operations, allowing for concurrent execution of multiple tasks. By using groups and communicators, you can organize processes to perform specific tasks independently.

11.4. Example: Task Parallelism

In this example, we divide processes into two groups: one for computing and one for I/O operations. Each group performs its respective task concurrently.

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

void perform_computation(int rank) {
    printf("Process %d performing computation\n", rank);
    // Simulate computation by sleeping for a while
    sleep(2);
}

void perform_io_operations(int rank) {
    printf("Process %d performing I/O operations\n", rank);
    // Simulate I/O by sleeping for a while
    sleep(3);
}

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

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


    // Split the world group into two groups based on rank
    int color = rank % 2;  // Determine color based on rank
    MPI_Comm new_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &new_comm);

    // Get the new rank and size in the new communicator
    int new_rank, new_size;
    MPI_Comm_rank(new_comm, &new_rank);
    MPI_Comm_size(new_comm, &new_size);

    if (color == 0) {
        perform_computation(rank);
    } else {
        perform_io_operations(rank);
    }

    // Free the new communicator and group
    MPI_Comm_free(&new_comm);
    MPI_Group_free(&world_group);

    MPI_Finalize();
    return 0;
}
```

11.5. Explanation

1. **Task Functions**:
   - `perform_computation` simulates a computation task.
   - `perform_io_operations` simulates an I/O task.
2. **Extract World Group**:
   - `MPI_Comm_group` is used to get the group of `MPI_COMM_WORLD`.
3. **Split the Communicator**:
   - `MPI_Comm_split` is used to split `MPI_COMM_WORLD` into two new communicators based on the color value (rank modulo 2).
   - This creates two new communicators: one for even ranks (compute group) and one for odd ranks (I/O group).
4. **Perform Task**:
   - Each group performs its respective task concurrently based on the rank's color.
5. **Cleanup**:
   - The new communicator and group are freed using `MPI_Comm_free` and `MPI_Group_free`.

11.6. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_task_parallelism.c
```

```
---------------------------------------------------------------
Command executed: mpicc mpi_task_parallelism.c -o mpi_task_parallelism.out
---------------------------------------------------------------
Compilation successful. Check at mpi_task_parallelism.out
---------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_task_parallelism.out 10
```

```
---------------------------------------------------------------
Command executed: mpirun -np 10 ./mpi_task_parallelism.out
---------------------------------------------------------------
###############################################################
##########                   OUTPUT                  ##########
###############################################################
```

```
##################################################################
##########                    DONE                    ##########
##################################################################
```

11.7. Communication between Groups

1. **Different Groups**:
    - **Inter-communicator**: MPI provides `MPI_Intercomm_create` to create an inter-communicator that allows communication between two different groups.
    - **Point-to-Point Communication**: `MPI_Send` and `MPI_Recv` can be used for direct communication between processes in different groups using the inter-communicator.
2. **Same Group**:
    - **Intra-communicator**: The default communicator within the same group is an intra-communicator (like `MPI_COMM_WORLD`). All standard communication operations (e.g., `MPI_Bcast`, `MPI_Reduce`) work within the same group.

11.8. Summary

Dividing processes into groups and communicators helps in organizing and managing tasks effectively in MPI. Different groups can communicate using inter-communicators, while communication within the same group uses intra-communicators.

- **Different Groups Communication**:
    - Use inter-communicators (`MPI_Intercomm_create`) and point-to-point communication (`MPI_Send`, `MPI_Recv`).
- **Same Group Communication**:
    - Use intra-communicators like `MPI_COMM_WORLD` and collective operations.

Groups and communicators do not share the same communicator. Each group can have its unique communicator, allowing for flexible and organized communication in parallel applications.

12. Task1

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

long long sumOfSquares(long long *arr, int size){
    long long sum = 0;
    for(int i = 0; i < size; i++){
        sum+= arr[i] * arr[i];
```

```
    }
    return sum;
}

long long sum(long long *arr, int size){
    long long sum = 0;
    for(int i = 0; i < size; i++){
        sum+= arr[i];
    }
    return sum;
}


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

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

    // Split the world group into two groups
    int color = rank % 2;  // Determine color based on rank
    MPI_Comm new_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &new_comm);

    // Get the new rank and size in the new communicator
    int new_rank, new_size;
    MPI_Comm_rank(new_comm, &new_rank);
    MPI_Comm_size(new_comm, &new_size);

    //printf("World Rank: %d, New Rank: %d, New Size: %d\n", rank, new_rank, new_size);
    const int data_size = 10000;
    long long data[data_size];
    if(new_rank == 0){
        for(int i = 0; i < data_size; i++){
            data[i] = i + 1;
        }
    }
    //data broadcasted to each process in new_comm
    int chunk_size = data_size / new_size;
    long long local_array[chunk_size];
    long long local_sum = 0;
    long long local_square_sum = 0;
    MPI_Scatter(data, chunk_size, MPI_LONG_LONG, local_array, chunk_size, MPI_LONG_LONG, 0, new_comm);
    // Perform some communication within the new communicator
    if(color == 0){
        local_sum = sum(local_array, chunk_size);
    }
    if(color == 1){
        local_square_sum = sumOfSquares(local_array, chunk_size);
    }
    long long final_sum = 0;
    long long final_square_sum = 0;
```

```c
    if(color == 0)
        MPI_Allreduce(&local_sum, &final_sum, 1, MPI_LONG_LONG, MPI_SUM, new_comm);
    else MPI_Allreduce(&local_square_sum, &final_square_sum, 1, MPI_LONG_LONG, MPI_SUM, new_comm);
    if(new_rank == 0){
        if(color == 0)
            printf("World Rank: %d, Sum of arrays: %lld\n", rank, final_sum);
        if(color == 1)
            printf("World Rank: %d, Sum of squares of arrays: %lld\n", rank, final_square_sum);
    }

    // Free the new communicator and group
    MPI_Comm_free(&new_comm);

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh taskp.c
```

```
----------------------------------------------------------------
Command executed: mpicc taskp.c -o taskp.out
----------------------------------------------------------------
Compilation successful. Check at taskp.out
----------------------------------------------------------------
```

```
bash run.sh ./taskp.out 10
```

```
----------------------------------------------------------------
Command executed: mpirun -np 10 ./taskp.out
----------------------------------------------------------------
################################################################
##########                    OUTPUT                  ##########
################################################################

World Rank: 1, Sum of squares of arrays: 333383335000
World Rank: 0, Sum of arrays: 50005000

################################################################
##########                     DONE                   ##########
################################################################
```

Author: Abhishek Raj
Created: 2025-01-13 Mon 17:27