Day16

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 Premitive Data Types

- MPI_CHAR
- MPI_WCHAR
- MPI_SHORT
- MPI_INT
- MPI_LONG
- MPI_LONG_LONG_INT
- MPI_LONG_LONG
- MPI_SIGNED_CHAR
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED_SHORT
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED
- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG_DOUBLE

3. MPI Derived Data Types

- Contiguous
- Vector
- Indexed
- Struct

4. MPI_Type_contiguous

`MPI_Type_contiguous` creates a new MPI datatype that represents a contiguous block of elements. This is useful when you want to send or receive a block of the same datatype as a single message.

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 20

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);
    }

    const int count = 5;
    int data[N];
    MPI_Datatype contiguous_type;

    // Create a contiguous datatype
    MPI_Type_contiguous(count, MPI_INT, &contiguous_type);
    MPI_Type_commit(&contiguous_type);

    if (rank == 0) {
        // Initialize the data array with some values
        for (int i = 0; i < N; i++) {
            data[i] = i + 1;
        }

        MPI_Send(data, 4, contiguous_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: ");
        for (int i = 0; i < N; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    } else if (rank == 1) {
        MPI_Recv(data, 4, contiguous_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: ");
        for (int i = 0; i < N; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

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

4.1. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_type_contiguous.c
```

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

- Run the program:

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

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

Process 0 sent data: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Process 1 received data: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

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

In this example, `MPI_Type_contiguous` is used to create a contiguous datatype that represents an array of integers. This datatype is then used to send and receive the array between processes.

5. MPI_Type_vector

`MPI_Type_vector` creates a new MPI datatype that represents a pattern of regularly spaced blocks of data. This is useful for sending or receiving non-contiguous data with a regular pattern, such as columns of a matrix or every nth element of an array.

5.1. Syntax

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

- `count`: Number of blocks.
- `blocklength`: Number of elements in each block.
- `stride`: Number of elements between the start of each block.
- `oldtype`: Datatype of each element in the block.
- `newtype`: New datatype representing the vector.

5.2. Example Code

```
#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);
    }
    const int count = 3;       // Number of blocks
    const int blocklength = 3; // Number of elements in each block
    const int stride = 6;      // Number of elements between the start of each block
    int data[15];              // Array to send/receive
    MPI_Datatype vector_type;
    MPI_Type_vector(count, blocklength, stride, MPI_INT, &vector_type);
    MPI_Type_commit(&vector_type);
    if (rank == 0) {
        for (int i = 0; i < 15; i++) {
            data[i] = i + 1;
        }

        MPI_Send(data, 1, vector_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: ");
        for (int i = 0; i < 15; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    } else if (rank == 1) {
        for (int i = 0; i < 15; i++) {
            data[i] = 0;
        }
        MPI_Recv(data, 1, vector_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: ");
        for (int i = 0; i < 15; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
```

```c
    }
    MPI_Type_free(&vector_type);
    MPI_Finalize();
    return 0;
}
```

5.3. Explanation

- **Initialization**: Initialize MPI, get the rank and size of the communicator.
- **Datatype Creation**: `MPI_Type_vector` creates a new datatype `vector_type` representing 3 blocks of 1 integer each, with a stride of 5 integers between the start of each block.
- **Process 0**:
    - Initializes the `data` array with values from 1 to 15.
    - Sends the `data` array using the `vector_type` to process 1.
    - Prints the `data` array.
- **Process 1**:
    - Initializes the `data` array to zero.
    - Receives the data from process 0 into the `data` array using the `vector_type`.
    - Prints the `data` array after receiving.
- **Datatype Cleanup**: Free the `vector_type` with `MPI_Type_free`.
- **Finalize**: Finalize the MPI environment.

5.4. Compilation and Execution

- Compile the program:

  ```bash
  bash compile.sh mpi_type_vector.c
  ```

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

- Run the program:

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

  ```
  ------------------------------------------------------------------
  Command executed: mpirun -np 2 ./mpi_type_vector.out
  ------------------------------------------------------------------
  ```

```
##################################################################
##########                    OUTPUT                    ##########
##################################################################

Process 0 sent data: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Process 1 received data: 1 2 3 0 0 0 7 8 9 0 0 0 13 14 15

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

This example demonstrates how to use `MPI_Type_vector` to communicate non-contiguous data with a regular pattern in MPI.

6. MPI_Type_vector Example2

```c
#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);
    }

    const int count = 3;       // Number of blocks
    const int blocklength = 2; // Number of elements in each block
    const int stride = 5;      // Number of elements between the start of each block
    MPI_Datatype vector_type;

    // Create a vector datatype
    MPI_Type_vector(count, blocklength, stride, MPI_INT, &vector_type);
    MPI_Type_commit(&vector_type);

    if (rank == 0) {
    int data[15];              // Array to send/receive
        // Initialize the data array with some values
        for (int i = 0; i < 15; i++) {
            data[i] = i + 1;
        }

        MPI_Send(data, 1, vector_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: ");
        for (int i = 0; i < 15; i++) {
```

```
            printf("%d ", data[i]);
        }
        printf("\n");
    } else if (rank == 1) {
        int data1[6];

        MPI_Recv(data1, count * blocklength, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: ");
        for (int i = 0; i < 6; i++) {
            printf("%d ", data1[i]);
        }
        printf("\n");
    }

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

6.1. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_type_vector1.c
```

```
------------------------------------------------------------------
Command executed: mpicc mpi_type_vector1.c -o mpi_type_vector1.out
------------------------------------------------------------------
Compilation successful. Check at mpi_type_vector1.out
------------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_type_vector1.out 2
```

```
------------------------------------------------------------------
Command executed: mpirun -np 2 ./mpi_type_vector1.out
------------------------------------------------------------------
##################################################################
##########                    OUTPUT                    ##########
##################################################################

Process 0 sent data: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Process 1 received data: 1 2 6 7 11 12

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

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

This example demonstrates how to use `MPI_Type_vector` to communicate non-contiguous data with a regular pattern in MPI.

7. MPI_Type_indexed

`MPI_Type_indexed` creates a new MPI datatype that represents an irregularly spaced set of blocks of data. This is useful for sending or receiving non-contiguous data with an irregular pattern.

7.1. Syntax

```
int MPI_Type_indexed(int count, const int array_of_blocklengths[], const int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newty
```

- `count`: Number of blocks.
- `array_of_blocklengths`: Array specifying the number of elements in each block.
- `array_of_displacements`: Array specifying the displacement of each block from the start.
- `oldtype`: Datatype of each element in the blocks.
- `newtype`: New datatype representing the indexed pattern.

7.2. Example Code

```
#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);
    }

    const int count = 3;
    int blocklengths[3] = {1, 2, 1};
    int displacements[3] = {0, 3, 7};
    int data[10];
    MPI_Datatype indexed_type;
```

```
    // Create an indexed datatype
    MPI_Type_indexed(count, blocklengths, displacements, MPI_INT, &indexed_type);
    MPI_Type_commit(&indexed_type);

    if (rank == 0) {
        // Initialize the data array with some values
        for (int i = 0; i < 10; i++) {
            data[i] = i + 1;
        }

        MPI_Send(data, 1, indexed_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: ");
        for (int i = 0; i < 10; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    } else if (rank == 1) {
        // Initialize the data array to zero
        for (int i = 0; i < 10; i++) {
            data[i] = 0;
        }

        MPI_Recv(data, 1, indexed_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: ");
        for (int i = 0; i < 10; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

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

7.3. Explanation

- **Initialization**: Initialize MPI, get the rank and size of the communicator.
- **Datatype Creation**:
    - `blocklengths` specifies the number of elements in each block: {1, 2, 1}.
    - `displacements` specifies the starting indices of each block: {0, 3, 7}.
    - `MPI_Type_indexed` creates a new datatype `indexed_type` representing these blocks.
- **Process 0**:
    - Initializes the `data` array with values from 1 to 10.
    - Sends the `data` array using the `indexed_type` to process 1.
    - Prints the `data` array.
- **Process 1**:
    - Initializes the `data` array to zero.

    - Receives the data from process 0 into the `data` array using the `indexed_type`.
    - Prints the `data` array after receiving.
- **Datatype Cleanup**: Free the `indexed_type` with `MPI_Type_free`.
- **Finalize**: Finalize the MPI environment.

7.4. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_type_indexed.c
```

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

- Run the program:

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

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

Process 0 sent data: 1 2 3 4 5 6 7 8 9 10
Process 1 received data: 1 0 0 4 5 0 0 8 0 0

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

This example demonstrates how to use `MPI_Type_indexed` to communicate non-contiguous data with an irregular pattern in MPI.

8. 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.

8.1. 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[],
```

8.2. 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.

8.3. 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;
    MPI_Datatype struct_type;

    // Create the datatype for my_struct
```

```c
    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;
}
```

8.4. 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.

8.5. 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.

9. 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;
```

```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] = {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;
}
```

9.1. 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
```

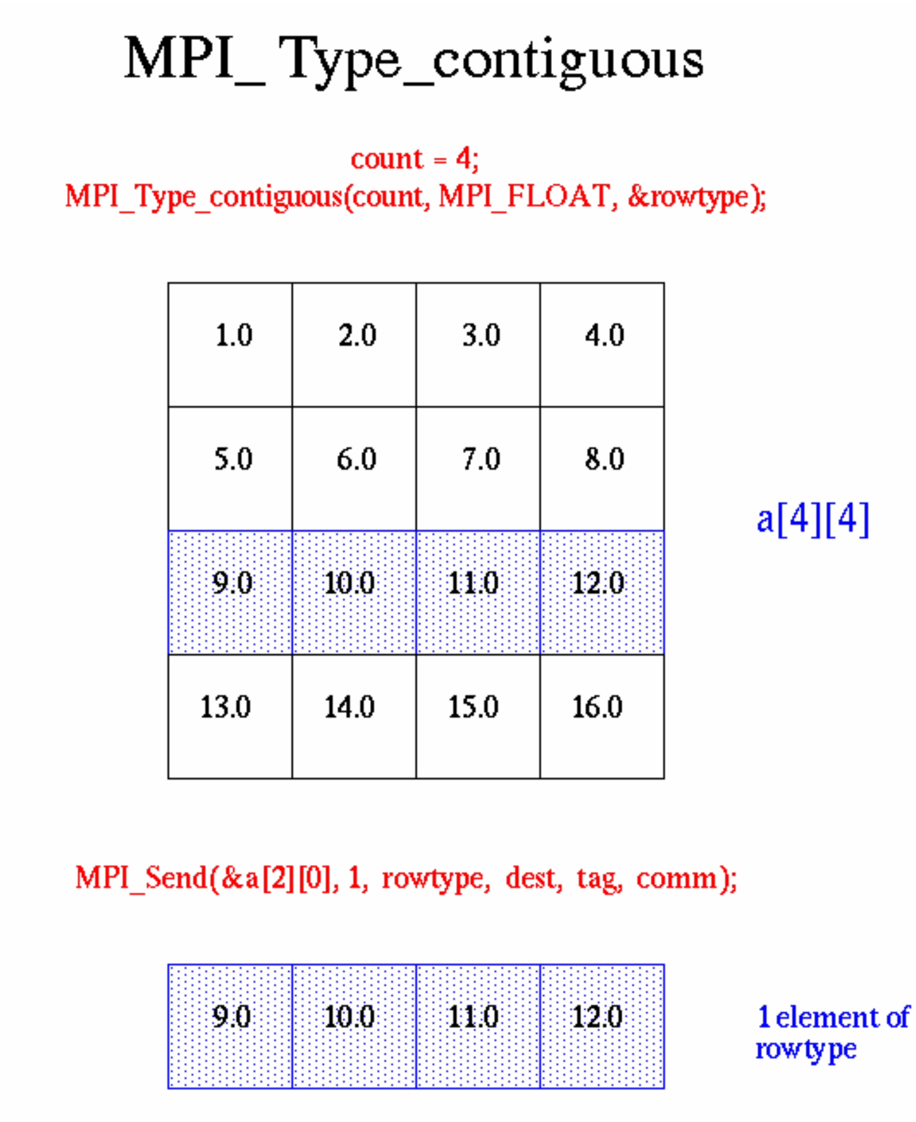
```
------------------------------------------------------------------
Command executed: mpirun -np 2 ./mpi_type_struct2.out
------------------------------------------------------------------
##################################################################
##########                    OUTPUT     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

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.

10. Reference

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

10.1. MPI Type contigous



10.2. MPI Type vector

MPI_ Type_vector

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

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

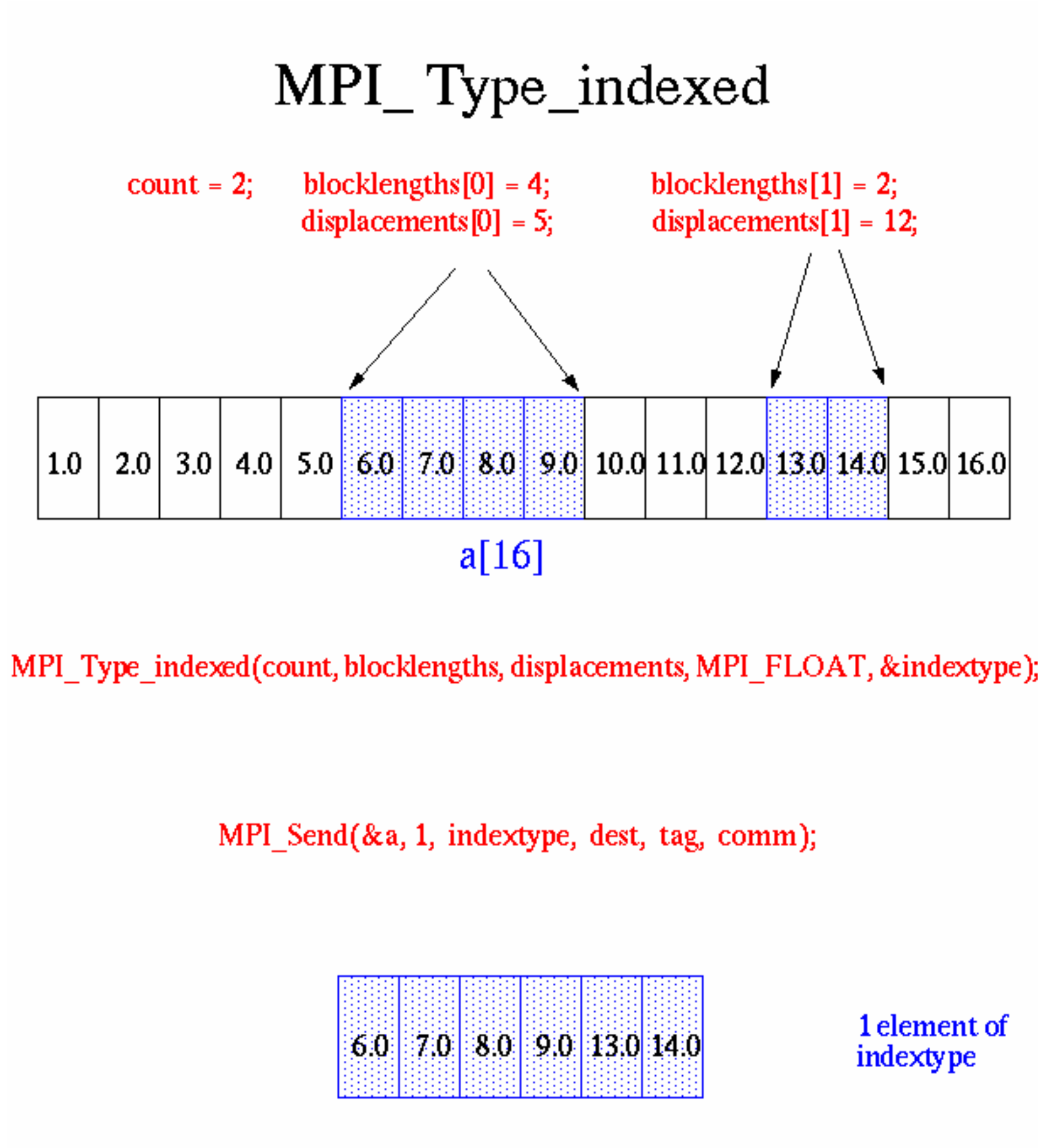
a[4][4]

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

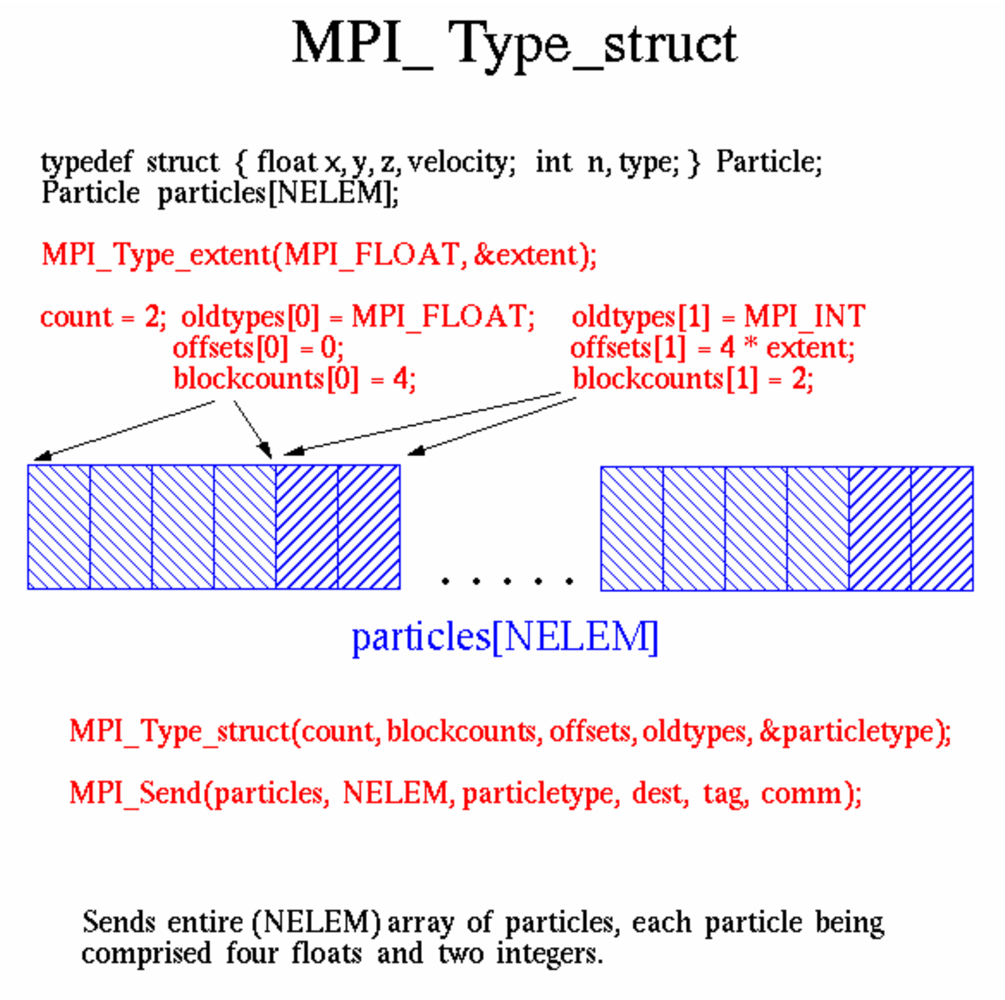
2.0	6.0	10.0	14.0

1 element of
columntype

10.3. MPI Type indexed



10.4. MPI Type struct



11. task1

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 5
```

```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);
    }
    int data[N];

    if (rank == 0) {
        // Initialize the data array with some values
        for (int i = 0; i < N; i++) {
            data[i] = i + 1;
        }

        MPI_Send(data, N, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: ");
        for (int i = 0; i < N; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    } else if (rank == 1) {
        MPI_Recv(data, N, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: ");
        for (int i = 0; i < N; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

11.1. 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 sent data: 1 2 3 4 5
Process 1 received data: 1 2 3 4 5

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

12. task2

Transfer a matrix of size N * N.

- Initialize a matrix for both process
- Allow only one process to create whole matrix
- That process will transfer the matrix to another process
- Print the matrix received.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 5

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);
```

```c
    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for this example\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    int data[N][N];

    if (rank == 0) {
        int temp = 1;
        // Initialize the data array with some values
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                data[i][j] = temp;
                temp++;
            }
        }

        MPI_Send(data, N * N, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: \n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%d\t", data[i][j]);
            }
            printf("\n");
        }
        printf("\n");
    } else if (rank == 1) {
        MPI_Recv(data, N * N, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: \n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%d\t", data[i][j]);
            }
            printf("\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

12.1. Compilation and Execution

- Compile the program:

```
bash compile.sh task2.c
```

```
-------------------------------------------------------------------
Command executed: mpicc task2.c -o task2.out
-------------------------------------------------------------------
Compilation successful. Check at task2.out
```

```
----------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./task2.out 2
```

```
----------------------------------------------------------------
Command executed: mpirun -np 2 ./task2.out
----------------------------------------------------------------
################################################################
##########                  OUTPUT                    ##########
################################################################

Process 0 sent data:
1      2       3       4       5
6      7       8       9       10
11     12      13      14      15
16     17      18      19      20
21     22      23      24      25

Process 1 received data:
1      2       3       4       5
6      7       8       9       10
11     12      13      14      15
16     17      18      19      20
21     22      23      24      25

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

13. task2 with MPI_Type_contiguous

Transfer a matrix of size N ∗ N.

- Initialize a matrix for both process
- Allow only one process to create whole matrix
- That process will transfer the matrix to another process
- Print the matrix received.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 5

int main(int argc, char** argv) {
```

```c
    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);
    }
    int data[N][N];
    MPI_Datatype contiguous_type;
    MPI_Type_contiguous(N, MPI_INT, &contiguous_type);
    MPI_Type_commit(&contiguous_type);

    if (rank == 0) {
        int temp = 1;
        // Initialize the data array with some values
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                data[i][j] = temp;
                temp++;
            }
        }

        MPI_Send(data[3], 2, contiguous_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: \n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%d\t", data[i][j]);
            }
            printf("\n");
        }
        printf("\n");
    } else if (rank == 1) {
        MPI_Recv(data[3], 2, contiguous_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: \n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%d\t", data[i][j]);
            }
            printf("\n");
        }
    }

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

13.1. Compilation and Execution

- Compile the program:

```
bash compile.sh task2_contiguous.c
```

```
------------------------------------------------------------------
Command executed: mpicc task2_contiguous.c -o task2_contiguous.out
------------------------------------------------------------------
Compilation successful. Check at task2_contiguous.out
------------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./task2_contiguous.out 2
```

```
------------------------------------------------------------------
Command executed: mpirun -np 2 ./task2_contiguous.out
------------------------------------------------------------------
##################################################################
##########                    OUTPUT                    ##########
##################################################################

Process 0 sent data:
1      2       3       4       5
6      7       8       9       10
11     12      13      14      15
16     17      18      19      20
21     22      23      24      25

Process 1 received data:
49152 0       -1630691416     32767   6
154    0       0       0       0
0      0       0       0       0
16     17      18      19      20
21     22      23      24      25

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

14. task3

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

```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);
    }
    const int count = 3;       // Number of blocks
    const int blocklength = 4; // Number of elements in each block
    const int stride = 6;      // Number of elements between the start of each block
    MPI_Datatype vector_type;
    MPI_Type_vector(count, blocklength, stride, MPI_INT, &vector_type);
    MPI_Type_commit(&vector_type);
    if (rank == 0) {
        int data[20];              // Array to send/receive
        for (int i = 0; i < 20; i++) {
            data[i] = i + 1;
        }
        MPI_Send(data, 1, vector_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: ");
        for (int i = 0; i < 20; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    } else if (rank == 1) {
        int size = count * blocklength;
        int data[size];
        MPI_Recv(data, size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }
    MPI_Type_free(&vector_type);
    MPI_Finalize();
    return 0;
}
```

14.1. Compilation and Execution

- Compile the program:

```
bash compile.sh task3.c
```

```
------------------------------------------------------------------
Command executed: mpicc task3.c -o task3.out
```

```
-----------------------------------------------------------------
Compilation successful. Check at task3.out
-----------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./task3.out 2
```

```
-----------------------------------------------------------------
Command executed: mpirun -np 2 ./task3.out
-----------------------------------------------------------------
#################################################################
##########                    OUTPUT                   ##########
#################################################################

Process 0 sent data: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Process 1 received data: 1 2 3 4 7 8 9 10 13 14 15 16

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

15. task4 (transfer 3rd column to process 1)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 8

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);
    }
    const int count = N;       // Number of blocks
    const int blocklength = 1; // Number of elements in each block
    const int stride = N;      // Number of elements between the start of each block
    MPI_Datatype vector_type;
    MPI_Type_vector(count, blocklength, stride, MPI_INT, &vector_type);
    MPI_Type_commit(&vector_type);
    int data[N][N];              // Array to send/receive
    if (rank == 0) {
        for (int i = 0; i < N; i++) {
```

```c
            for(int j = 0; j < N; j++){
                data[i][j] = i + 1;
            }
        }
        MPI_Send(&data[0][2], 1, vector_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: \n");
        for (int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++){
                printf("%d ", data[i][j]);
            }
            printf("\n");
        }
        printf("\n");
    } else if (rank == 1) {
        for (int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++){
                data[i][j] = 0;
            }
        }
        MPI_Recv(&data[0][2], 1, vector_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: \n");
        for (int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++){
                printf("%d ", data[i][j]);
            }
            printf("\n");
        }
        printf("\n");
    }
    MPI_Type_free(&vector_type);
    MPI_Finalize();
    return 0;
}
```

15.1. Compilation and Execution

- Compile the program:

```
bash compile.sh task4.c
```

```
-------------------------------------------------------------------
Command executed: mpicc task4.c -o task4.out
-------------------------------------------------------------------
Compilation successful. Check at task4.out
-------------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./task4.out 2
```

```
-----------------------------------------------------------------
Command executed: mpirun -np 2 ./task4.out
-----------------------------------------------------------------
#################################################################
##########                    OUTPUT                   ##########
#################################################################

Process 0 sent data:
1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8

Process 1 received data:
0 0 1 0 0 0 0 0
0 0 2 0 0 0 0 0
0 0 3 0 0 0 0 0
0 0 4 0 0 0 0 0
0 0 5 0 0 0 0 0
0 0 6 0 0 0 0 0
0 0 7 0 0 0 0 0
0 0 8 0 0 0 0 0


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

16. Implement vector using indexed

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 20

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);
    }

    const int count = 4;
    const int sl = 5;                  //sl = stridelength
    int blocklengths[4] = {2, 2, 2, 2};
    int displacements[4] = {0 * sl, 1 * sl, 2 * sl, 3 * sl};
    //int displacements[3] = {0, 5, 10};
    int data[N];
    MPI_Datatype indexed_type;

    // Create an indexed datatype
    MPI_Type_indexed(count, blocklengths, displacements, MPI_INT, &indexed_type);
    MPI_Type_commit(&indexed_type);

    if (rank == 0) {
        // Initialize the data array with some values
        for (int i = 0; i < N; i++) {
            data[i] = i + 1;
        }

        MPI_Send(data, 1, indexed_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: ");
        for (int i = 0; i < N; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    } else if (rank == 1) {
        // Initialize the data array to zero
        for (int i = 0; i < N; i++) {
            data[i] = 0;
        }

        MPI_Recv(data, 1, indexed_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: ");
        for (int i = 0; i < N; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

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

16.1. Compilation and Execution

- Compile the program:

```
bash compile.sh task5.c
```

```
-----------------------------------------------------------------
Command executed: mpicc task5.c -o task5.out
-----------------------------------------------------------------
Compilation successful. Check at task5.out
-----------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./task5.out 2
```

```
-----------------------------------------------------------------
Command executed: mpirun -np 2 ./task5.out
-----------------------------------------------------------------
##################################################################
##########                    OUTPUT                    ##########
##################################################################

Process 0 sent data: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Process 1 received data: 1 2 0 0 0 6 7 0 0 0 11 12 0 0 0 16 17 0 0 0

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

17. Implement contiguous using indexed

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 20

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);
    }
```

```
    const int count = 4;
    const int sl = 5;                  //sl = stridelength
    int blocklengths[4] = {sl, sl, sl, sl};     //need to have stride length equal to block length to make it contiguous
    int displacements[4] = {0 * sl, 1 * sl, 2 * sl, 3 * sl};
    //int displacements[3] = {0, 5, 10};
    int data[N];
    MPI_Datatype indexed_type;

    // Create an indexed datatype
    MPI_Type_indexed(count, blocklengths, displacements, MPI_INT, &indexed_type);
    MPI_Type_commit(&indexed_type);

    if (rank == 0) {
        // Initialize the data array with some values
        for (int i = 0; i < N; i++) {
            data[i] = i + 1;
        }

        MPI_Send(data, 1, indexed_type, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data: ");
        for (int i = 0; i < N; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    } else if (rank == 1) {
        // Initialize the data array to zero
        for (int i = 0; i < N; i++) {
            data[i] = 0;
        }

        MPI_Recv(data, 1, indexed_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: ");
        for (int i = 0; i < N; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

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

17.1. Compilation and Execution

- Compile the program:

```
bash compile.sh task6.c
```

```
-------------------------------------------------------------------
```

```
Command executed: mpicc task6.c -o task6.out
-----------------------------------------------------------------
Compilation successful. Check at task6.out
-----------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./task6.out 2
```

```
-----------------------------------------------------------------
Command executed: mpirun -np 2 ./task6.out
-----------------------------------------------------------------
#################################################################
##########                    OUTPUT                   ##########
#################################################################

Process 0 sent data: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Process 1 received data: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

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

Author: Abhishek Raj
Created: 2025-01-13 Mon 17:23