Day7

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 with different blocklength

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

typedef struct {
    int arr[3];
    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] = {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, da
    }

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

2.1. 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.

3. 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.

3.1. Functions

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

3.2. 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);
```

3.3. 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;
```

```c
        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, position, 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;
}
```

3.4. 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.

3.5. 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.

4. MPI Pack Size, Probe, and Get Count

4.1. 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.

4.1.1. 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).

4.1.2. 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                     ##########
##################################################################
```

4.2. 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.

4.2.1. 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).

4.3. 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.

4.3.1. 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).

4.4. 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);
```

```c
    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;
}
```

4.5. 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.

4.6. 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                     ##########
```

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

5. 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 5

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

6. Task v2

```
#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);
```

```c
        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 8

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


Author: Abhishek Raj

Created: 2024-07-13 Sat 07:38