Day11

Table of Contents

1. Agenda

- Recap
- Communicators
- MPI Communications
- MPI Programs
- Point to point communications
- Synchronous and Asynchronous calls
- Non-blocking calls
- Sum using p2p communication

2. Scripts

2.1. compile script

```sh
#!/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 "-----------------------------------------------------------------"
```

2.2. run script

```sh
#!/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 "#################################################################"
```

3. MPI Communicators

Communicators in MPI define a group of processes that can communicate with each other. The default communicator is `MPI_COMM_WORLD`, which includes all the processes. Custom communicators can be created to define subgroups of processes for specific communication patterns.

4. Types of MPI Communications

MPI offers various communication mechanisms to facilitate different types of data exchanges between processes:

4.1. Point-to-Point Communication:

- **Blocking**: The sending and receiving operations wait until the message is delivered (e.g., `MPI_Send`, `MPI_Recv`).
- **Non-Blocking**: The operations return immediately, allowing computation and communication to overlap (e.g., `MPI_Isend`, `MPI_Irecv`).

4.2. Collective Communication:

These operations involve a group of processes and include:

- **Broadcast**: Send data from one process to all other processes (`MPI_Bcast`).
- **Scatter**: Distribute distinct chunks of data from one process to all processes (`MPI_Scatter`).
- **Gather**: Collect chunks of data from all processes to one process (`MPI_Gather`).
- **All-to-All**: Every process sends and receives distinct chunks of data (`MPI_Alltoall`).

Collectives can also include operations like reductions (`MPI_Reduce`, `MPI_Allreduce`) which perform computations on data from all processes and distribute the result.

5. Point-to-point communication

```
#include"stdio.h"
#include"mpi.h"

int main(int argc, char **argv)
{
    int myid, size;
    int myval;

    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    if(myid==0){
        myval = 100;
        printf("\nmyid: %d \t myval = %d", myid, myval);
        MPI_Send(&myval, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("\nmyid: %d \t Data sent.\n", myid);
    }
    else if(myid==1){   // Process with ID exactly equal to 1
        myval = 200;
        MPI_Recv(&myval, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("\nmyid: %d \t Data received.", myid);
        printf("\nmyid: %d \t myval = %d", myid, myval);
        printf("\n\nProgram exit!\n");
    }
```

```
    //End MPI environment
    MPI_Finalize();
}
```

```
bash compile.sh p2p_mpi.c
```

```
------------------------------------------------------------------
Command executed: mpicc p2p_mpi.c -o p2p_mpi.out
------------------------------------------------------------------
Compilation successful. Check at p2p_mpi.out
------------------------------------------------------------------
```

```
bash run.sh ./p2p_mpi.out 2
```

```
------------------------------------------------------------------
Command executed: mpirun -np 2 ./p2p_mpi.out
------------------------------------------------------------------
##################################################################
##########                    OUTPUT                    ##########
##################################################################


myid: 0          myval = 100
myid: 0          Data sent.

myid: 1          Data received.
myid: 1          myval = 100

Program exit!

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

5.1. Sending array to process 1

```
#include"stdio.h"
```

```c
#include"mpi.h"
#define N 100

int main()
{
    int myid, size;
    int myval;

    int arr[N];
    //Initialize MPI environment
    MPI_Init(NULL,NULL);

    //Get total number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //Get my unique ID among all processes
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    // Process with ID exactly equal to 0
    if(myid==0){
        //Initialize data to be sent
        for(int i = 0; i < N; i++) arr[i] = i + 1;
        //Send data
        MPI_Send(arr, N, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("\nmyid: %d \t Data sent.\n", myid);
    }
    else if(myid==1){   // Process with ID exactly equal to 1
        //Initialize receive array to some other data
        MPI_Recv(arr, N, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("\nmyid: %d \t Data received.\n", myid);
        //Print received data
        for(int i = 0; i < N; i++)
          printf("%d ", arr[i]);
    }

    //End MPI environment
    MPI_Finalize();
}
```

```
bash compile.sh p2p_mpi_array.c
```

```
------------------------------------------------------------------
Command executed: mpicc p2p_mpi_array.c -o p2p_mpi_array.out
------------------------------------------------------------------
```

```
Compilation successful. Check at p2p_mpi_array.out
------------------------------------------------------------------
```

```
bash run.sh ./p2p_mpi_array.out 2
```

```
------------------------------------------------------------------
Command executed: mpirun -np 2 ./p2p_mpi_array.out
------------------------------------------------------------------
##################################################################
##########                    OUTPUT                    ##########
##################################################################


myid: 0          Data sent.

myid: 1          Data received.
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 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
##################################################################
##########                     DONE                     ##########
##################################################################
```

6. MPI Communication: Synchronous and Asynchronous

6.1. Synchronous Communication using MPI_Send and MPI_Recv

In synchronous communication, the send operation does not complete until the matching receive operation has been started.

6.1.1. mpi_sync.c

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

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

    int rank;
```

```c
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size;
    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 number;
    if (rank == 0) {
        number = -1;
        MPI_Ssend(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent number %d to process 1\n", number);
    } else if (rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", number);
    }

    MPI_Finalize();
    return 0;
}
```

6.1.2. Compilation and Execution (Synchronous)

- Compile the program:

```
bash compile.sh mpi_sync.c
```

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

- Run the program:

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

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

Process 1 received number -1 from process 0
Process 0 sent number -1 to process 1

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

6.2. Asynchronous Communication using MPI_Isend and MPI_Irecv

In asynchronous communication, the send operation can complete before the matching receive operation starts. Non-blocking operations allow computation and communication to overlap.

6.2.1. mpi_async.c

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

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

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size;
    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 number;
    if (rank == 0) {
        number = -1;
        MPI_Request request;
```

```c
        MPI_Isend(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent number %d to process 1\n", number);
    } else if (rank == 1) {
        MPI_Request request;
        MPI_Irecv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        printf("Process 1 received number %d from process 0\n", number);
    }

    MPI_Finalize();
    return 0;
}
```

6.2.2. Compilation and Execution (Asynchronous)

- Compile the program:

```
bash compile.sh mpi_async.c
```

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

- Run the program:

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

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

Process 0 sent number -1 to process 1
Process 1 received number 29908 from process 0
```

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

7. MPI Array Sum Calculation Example

7.1. flow of your program

- initialize mpi environment
- let process 0 create and initialize the whole data
- now process 0 will send the complete data to all other process
- now every process is having the complete data
- to define start and end for each process to allow them perform computation on their part of data only
- every process will start their computation of performing localsum on their part of data from start to end
- then each process will send their computed localsum to process 0
- 0 should receive the localsum of each process and at the same time it should add it to a variable totalsum
- once your total localsum is received by all the process 0 should print the result on your screen.
- finalize mpi environment

7.2. mpi_array_sum.c

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

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

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int n = 10000; // Size of the array
```

```c
int *array = NULL;
int chunk_size = n / world_size;
int *sub_array = (int*)malloc(chunk_size * sizeof(int));

if (world_rank == 0) {
    array = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        array[i] = i + 1; // Initialize the array with values 1 to n
    }

    // Distribute chunks of the array to other processes
    for (int i = 1; i < world_size; i++) {
        MPI_Send(array + i * chunk_size, chunk_size, MPI_INT, i, 0, MPI_COMM_WORLD);
    }

    // Copy the first chunk to sub_array
    for (int i = 0; i < chunk_size; i++) {
        sub_array[i] = array[i];
    }
} else {
    // Receive chunk of the array
    MPI_Recv(sub_array, chunk_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

// Compute the local sum
int local_sum = 0;
for (int i = 0; i < chunk_size; i++) {
    local_sum += sub_array[i];
}

if (world_rank != 0) {
    // Send local sum to process 0
    MPI_Send(&local_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
} else {
    // Process 0 receives the local sums and computes the final sum
    int final_sum = local_sum;
    int temp_sum;
    for (int i = 1; i < world_size; i++) {
        MPI_Recv(&temp_sum, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        final_sum += temp_sum;
    }
    printf("The total sum of array elements is %d\n", final_sum);
}

free(sub_array);
if (world_rank == 0) {
    free(array);
```

```
    }

    MPI_Finalize();
    return 0;
}
```

7.3. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_array_sum.c
```

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

- Run the program:

```
bash run.sh ./mpi_array_sum.out 7
```

```
------------------------------------------------------------------
Command executed: mpirun -np 7 ./mpi_array_sum.out
------------------------------------------------------------------
##################################################################
##########                   OUTPUT                    ##########
##################################################################

The total sum of array elements is 49965006

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

8. Task1

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

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

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size;
    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 number;
    if (rank == 0) {
        number = 100;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent number %d to process 1\n", number);
        MPI_Recv(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 0 received number %d from process 1\n", number);
    } else if (rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", number);
        number = 200;
        MPI_Send(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        printf("Process 1 sent number %d to process 0\n", number);
    } else{
        printf("I am process %d and I have nothing to do\n", rank);
    }

    MPI_Finalize();
    return 0;
}
```

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

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

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

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

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

Process 0 sent number 100 to process 1
Process 0 received number 200 from process 1
Process 1 received number 100 from process 0
Process 1 sent number 200 to process 0

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

9. Task2

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

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

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size;
    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 number1, number2;
    if (rank == 0) {
        number1 = 100;
```

```c
        number2 = 200;
        MPI_Send(&number1, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&number2, 1, MPI_INT, 1, 2, MPI_COMM_WORLD);
        printf("Process 0 sent number %d to process 1\n", number1);
        printf("Process 0 sent number %d to process 1\n", number2);
    } else if (rank == 1) {
        MPI_Recv(&number1, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&number2, 1, MPI_INT, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 n1 received number %d from process 0\n", number1);
        printf("Process 1 n2 received number %d from process 0\n", number2);
    } else{
        printf("I am process %d and I have nothing to do\n", rank);
    }

    MPI_Finalize();
    return 0;
}
```

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

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

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

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

Process 0 sent number 100 to process 1
Process 0 sent number 200 to process 1
Process 1 n1 received number 100 from process 0
Process 1 n2 received number 200 from process 0

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

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

10. Task3

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

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

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size;
    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 number1, number2;
    if (rank == 0) {
        number1 = 100;
        number2 = 200;
        MPI_Request request;
        MPI_Isend(&number1, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
        MPI_Isend(&number2, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
        printf("Process 0 sent number %d to process 1\n", number1);
        printf("Process 0 sent number %d to process 1\n", number2);
    } else if (rank == 1) {
        MPI_Request request;
        MPI_Irecv(&number1, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        MPI_Irecv(&number2, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", number1);
        printf("Process 1 received number %d from process 0\n", number2);
    } else{
        printf("I am process %d and I have nothing to do\n", rank);
    }

    MPI_Finalize();
    return 0;
```

```
}
```

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

```
------------------------------------------------------------------
Command executed: mpicc task3.c -o task3.out
------------------------------------------------------------------
Compilation successful. Check at task3.out
------------------------------------------------------------------
```

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

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

Process 0 sent number 100 to process 1
Process 0 sent number 200 to process 1
Process 1 received number 100 from process 0
Process 1 received number 200 from process 0

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

11. Task4

```
#include <mpi.h>
#include <stdio.h>
#define N 10000

int main(int argc, char** argv) {
    int arr[N];
    for(int i = 0; i < N; i++){
        arr[i] = i + 1;
```

```c
    }
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int chunksize = N / size;
    int start = chunksize * rank;
    int end = (rank + 1) * chunksize;
    if(rank == size - 1) end = N;
    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for this example\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    int localSum = 0;
    for(int i = start; i < end; i++){
        localSum+= arr[i];
    }
    if(rank != 0){
        MPI_Send(&localSum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    if (rank == 0) {
        int totalSum = 0;
        totalSum += localSum;
        for(int i = 1; i < size; i++){
            MPI_Recv(&localSum, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            totalSum += localSum;
        }
        printf("Total sum = %d\n", totalSum);
    }

    MPI_Finalize();
    return 0;
}
```

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

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

```
bash run.sh ./task4.out 10
```

```
------------------------------------------------------------------
Command executed: mpirun -np 10 ./task4.out
------------------------------------------------------------------
##################################################################
##########                    OUTPUT                    ##########
##################################################################

Total sum = 50005000

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

12. test

12.1. test.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);

    int n = 10000; // Size of the array
    int *arr = NULL;
    int localsum = 0;

    if (rank == 0) {
        // Allocate and initialize the array
        arr = (int*)malloc(sizeof(int) * n);
        for (int i = 0; i < n; i++) {
            arr[i] = i + 1;
        }
```

```
    // Distribute chunks of the array to other processes
    int chunksize = n / size;
    for (int i = 1; i < size; i++) {
        int start = i * chunksize;
        int end = (i == size - 1) ? n : start + chunksize;
        int send_size = end - start;

        MPI_Send(&arr[start], send_size, MPI_INT, i, 0, MPI_COMM_WORLD);
    }

    // Calculate the local sum for rank 0's chunk
    for (int i = 0; i < chunksize; i++) {
        localsum += arr[i];
    }

    // Receive local sums from other processes and compute total sum
    int totalsum = localsum;
    for (int i = 1; i < size; i++) {
        MPI_Recv(&localsum, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        totalsum += localsum;
    }

    // Print the total sum
    printf("Total sum = %d\n", totalsum);

    // Free the array
    free(arr);
} else {
    // Calculate chunksize and allocate a buffer for received data
    int chunksize = n / size;
    int start = rank * chunksize;
    int end = (rank == size - 1) ? n : start + chunksize;
    int recv_size = end - start;

    int *recv_buf = (int*)malloc(sizeof(int) * recv_size);

    // Receive the chunk from rank 0
    MPI_Recv(recv_buf, recv_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Calculate the local sum
    for (int i = 0; i < recv_size; i++) {
        localsum += recv_buf[i];
    }

    // Send the local sum back to rank 0
    MPI_Send(&localsum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

```
        // Free the buffer
        free(recv_buf);
    }

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh test.c
```

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

```
bash run.sh ./test.out 8
```

```
-----------------------------------------------------------------
Command executed: mpirun -np 8 ./test.out
-----------------------------------------------------------------
##################################################################
##########                   OUTPUT                    ##########
##################################################################

Total sum = 50005000

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

13. sum1.c

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

```c
int main(){
    int size, rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    const int n = 1000;
    int arr[n];
    for(int i = 0; i < n; i++) arr[i] = i + 1;
    int chunksize = n / size;
    int start = rank * chunksize;
    int end = start + chunksize;
    if(rank == size - 1){
        end = n;
    }

    int localsum = 0;
    for(int i = start; i < end; i++){
        localsum += arr[i];
    }

    if(rank != 0){
        MPI_Send(&localsum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else{
        int totalsum = 0;
        totalsum += localsum;
        for(int i = 1; i < size; i++){
            MPI_Recv(&localsum, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            totalsum += localsum;
        }
        printf("totalsum = %d\n", totalsum);
    }

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh sum1.c
```

```
-----------------------------------------------------------------
Command executed: mpicc sum1.c -o sum1.out
-----------------------------------------------------------------
Compilation successful. Check at sum1.out
-----------------------------------------------------------------
```

```
bash run.sh ./sum1.out 7
```

```
-----------------------------------------------------------------
Command executed: mpirun -np 7 ./sum1.out
-----------------------------------------------------------------
#################################################################
##########                   OUTPUT                   ##########
#################################################################

totalsum = 500500

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

14. sum2.c

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
int main(){
    int size, rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    const int n = 10000;
    int arr[n];
    if(rank == 0){
        for(int i = 0; i < n; i++) arr[i] = i + 1;
        for(int i = 1; i < size; i++){
            MPI_Send(arr, n, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    }
    else{
```

```
        MPI_Recv(arr, n, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    int chunksize = n / size;
    int start = rank * chunksize;
    int end = start + chunksize;
    if(rank == size - 1){
        end = n;
    }

    int localsum = 0;
    for(int i = start; i < end; i++){
        localsum += arr[i];
    }

    if(rank != 0){
        MPI_Send(&localsum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else{
        int totalsum = 0;
        totalsum += localsum;
        for(int i = 1; i < size; i++){
            MPI_Recv(&localsum, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            totalsum += localsum;
        }
        printf("totalsum = %d\n", totalsum);
    }

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh sum2.c
```

```
------------------------------------------------------------------
Command executed: mpicc sum2.c -o sum2.out
------------------------------------------------------------------
Compilation successful. Check at sum2.out
------------------------------------------------------------------
```

```
bash run.sh ./sum2.out 7
```

```
-----------------------------------------------------------------
Command executed: mpirun -np 7 ./sum2.out
-----------------------------------------------------------------
#################################################################
##########                  OUTPUT                     ##########
#################################################################

totalsum = 50005000

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

15. sum3.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
int main(){
    int size, rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    const int n = 1000;
    int arr[n];
    int chunksize = n / size;
    int start = rank * chunksize;
    int end = start + chunksize;
    if(rank == size - 1){
        end = n;
    }
    for(int i = start; i < end; i++){
        arr[i] = i + 1;
    }

    int localsum = 0;
    for(int i = start; i < end; i++){
        localsum += arr[i];
    }

    if(rank != 0){
        MPI_Send(&localsum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
```

```c
    else{
        int totalsum = 0;
        totalsum += localsum;
        for(int i = 1; i < size; i++){
            MPI_Recv(&localsum, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            totalsum += localsum;
        }
        printf("totalsum = %d\n", totalsum);
    }

    MPI_Finalize();
    return 0;
}
```

```bash
bash compile.sh sum3.c
```

```
-------------------------------------------------------------------
Command executed: mpicc sum3.c -o sum3.out
-------------------------------------------------------------------
Compilation successful. Check at sum3.out
-------------------------------------------------------------------
```

```bash
bash run.sh ./sum3.out 6
```

```
-------------------------------------------------------------------
Command executed: mpirun -np 6 ./sum3.out
-------------------------------------------------------------------
###################################################################
##########                    OUTPUT                     ##########
###################################################################

totalsum = 500500

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

Author: Abhishek Raj
Created: 2025-01-03 Fri 12:11