

Name: Manthan Gopal Dhole

ID : 201080020

Sub : Parallel Computing

Parallel Computing experiment 7

Aim:

To implement following using MPI.

- i. calculating Rank and Number of processor.**
- ii. Pi calculation.**
- iii. advanced MPI program that has a total number of 4 processes, where the process with rank = 0 should send VJTI letter to all the processes using MPI_Scatter call.**
- iv. find the maximum value in array of six integers with 6 processes, and print the result in root process using MPI_Reduce call.**
- v. Ring topology.**

Theory:

MPI

MPI (Message Passing Interface) is a standardized communication protocol that allows multiple processes running on different computing nodes to exchange messages and data. It is commonly used in parallel and distributed computing to implement communication between processes, especially in high-performance computing (HPC) applications.

MPI provides a set of functions and data types that enable processes to send and receive messages with each other, synchronize their activities, and coordinate their computations. It supports various communication modes, such as point-to-point, collective, and one-sided communication, and can be used with different programming languages, including C, C++, Fortran, and Python.

MPI has several implementations available, including MPICH, Open MPI, and Intel MPI, and is widely used in scientific, engineering, and data analysis applications that require parallel processing on multiple processors or computing clusters.

i. calculating Rank and Number of processor.

In MPI, each process is assigned a unique identifier called rank, which ranges from 0 to the total number of processes running in the MPI job. The rank is used to distinguish one process from another and to determine which process should receive or send messages.

To obtain the rank of the current process and the total number of processes in the MPI job, you can use the `MPI_Comm_rank()` and `MPI_Comm_size()` functions, respectively.

ii. Pi calculation.

Calculating Pi using MPI is a common parallel programming exercise that demonstrates the use of MPI for distributing computation across multiple processes.

Our program calculates an approximation of pi using the Monte Carlo method. The program distributes the computation of the sum of a series across multiple MPI processes and then combines the partial results to obtain the final value of pi.

iii. advanced MPI program that has a total number of 4 processes, where the process with rank = 0 should send VJTI letter to all the processes using `MPI_Scatter` call.

In Our program, the message "VJTI" is stored in the msg array in process 0. The MPI_Scatter call is used to send a part of this message to each of the other processes. The message is split into 4 character chunks and each process receives one chunk. The message is received into the same msg array to simplify the code, but in a real program you would probably use a separate receive buffer.

When each process receives its part of the message, it prints a message to the console indicating the rank of the process and the part of the message it received.

iv. find the maximum value in array of six integers with 6 processes, and print the result in root process using MPI_Reduce call.

In our program, each process initializes a local variable local_max with the value of the corresponding element in the array. The MPI_Reduce call is used to find the maximum value across all processes using the MPI_MAX operation. The result is stored in the max_value variable on the root process. Finally, the root process prints the maximum value.

Note that in a real program, the array would probably be distributed among the processes using MPI_Scatter or a similar function, but for simplicity we've just used a hardcoded array here.

v. Ring topology.

In this program, each process sends a message to its right neighbor and receives a message from its left neighbor. The rank of each neighbor is calculated using the modulo operator (%) to ensure that the indices wrap around in a circular fashion. The MPI_Send and MPI_Recv functions are used for communication, with the appropriate ranks of the neighbors passed as arguments. Finally, each process prints the message it received.

Note that in a real program, you would typically use a loop to pass the message around the ring multiple times, or to pass different messages with different tags or data types.

Code link:

<https://colab.research.google.com/drive/1J3EFKJCQBw6DFLLeguWPpRbkNnbLrgEH?usp=sharing>

Code and Output:

i. calculating Rank and Number of processor.

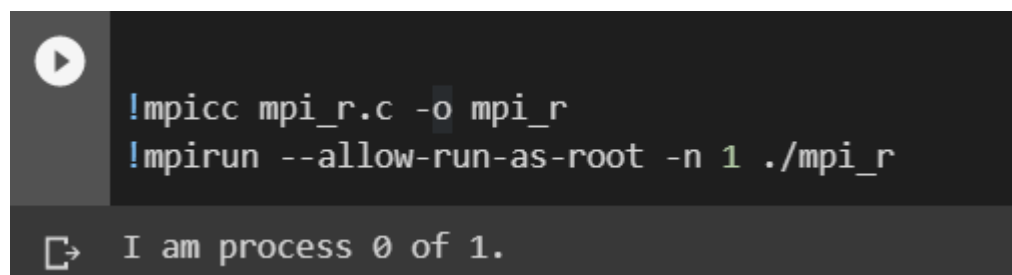
```
%%writefile mpi_r.c
#include <mpi.h>
#include <stdio.h>

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

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("I am process %d of %d.\n", rank, size);

    MPI_Finalize();
    return 0;
}
```



```
!mpicc mpi_r.c -o mpi_r
!mpirun --allow-run-as-root -n 1 ./mpi_r

I am process 0 of 1.
```

ii. Pi calculation.

```
%%writefile mpi_pi.c
#include <mpi.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char** argv) {
    int rank, size;
```

```

long long int n = 1000000;
double x, pi, sum = 0.0;
int i;

MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

for (i = rank; i < n; i += size) {
    x = (double)(i + 0.5) / (double)n;
    sum += 4.0 / (1.0 + x * x);
}

double local_pi = sum / n;
MPI_Reduce(&local_pi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)
;

if (rank == 0) {
    printf("pi is approximately %.16f.\n", pi);
}

MPI_Finalize();
return 0;
}

```

```

[ ] !mpicc mpi_pi.c -o mpi_pi
    !mpirun --allow-run-as-root -np 1 ./mpi_pi

```

```

pi is approximately 3.1415926535897643.

```

iii. advanced MPI program that has a total number of 4 processes, where the process with rank = 0 should send VJTI letter to all the processes using MPI_Scatter call.

```

%%writefile mpi_vjti.c
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define MESSAGE_SIZE 5

```

```

int main(int argc, char** argv) {
    int rank, size;
    char message[MESSAGE_SIZE + 1];

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        strcpy(message, "VJTI");
    }

    MPI_Scatter(message, MESSAGE_SIZE, MPI_CHAR, message, MESSAGE_SIZE, MPI_CHAR, 0, MPI_COMM_WORLD);

    printf("Process %d received message: %s\n", rank, message);

    MPI_Finalize();
    return 0;
}

```

```

!mpicc mpi_vjti.c -o mpi_vjti
!mpirun --allow-run-as-root -np 1 ./mpi_vjti

```

Process 0 received message: VJTI

iv. find the maximum value in array of six integers with 6 processes, and print the result in root process using MPI_Reduce call.

```

%%writefile mpi_maximum.c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank, size, max_value = 0;
    const int root = 0;
    int arr[6] = {5, 10, 7, 3, 8, 1};
    int local_max = arr[rank];

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

```

```

MPI_Comm_size(MPI_COMM_WORLD, &size);

MPI_Reduce(&local_max, &max_value, 1, MPI_INT, MPI_MAX, root, MPI_COMM_WORLD);

if (rank == root) {
    printf("The maximum value is %d.\n", max_value);
}


MPI_Finalize();
return 0;
}

```

```

!mpicc mpi_maximum.c -o mpi_maximum
!mpirun --allow-run-as-root -np 1 ./mpi_maximum

```

 The maximum value in the array is: 1

v. Ring topology.

```

%%writefile mpi_ring.c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank, size;
    int message_in, message_out;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        message_out = 1;
        MPI_Send(&message_out, 1, MPI_INT, (rank + 1) % size, 0, MPI_COMM_WORLD);
        MPI_Recv(&message_in, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process %d received message %d from process %d.\n", rank, message_in, size - 1);
    } else {

```

```
MPI_Recv(&message_in, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_
STATUS_IGNORE);
printf("Process %d received message %d from process %d.\n", rank, m
essage_in, rank - 1);
if (rank == size - 1) {
    message_out = 0;
} else {
    message_out = message_in + 1;
}
MPI_Send(&message_out, 1, MPI_INT, (rank + 1) % size, 0, MPI_COMM_W
ORLD);
}

MPI_Finalize();
return 0;
}
```



```
!mpicc mpi_ring.c -o mpi_ring
!mpirun --allow-run-as-root -np 1 ./mpi_ring
```



```
Process 0 received message 1 from process 0.
```

Conclusion:

We have Implemented some operations in MPI. In this experiment, we have learned deeply about the Message Passing Interface. MPI programs typically divide the problem into smaller subproblems, which are solved concurrently by different processes.