

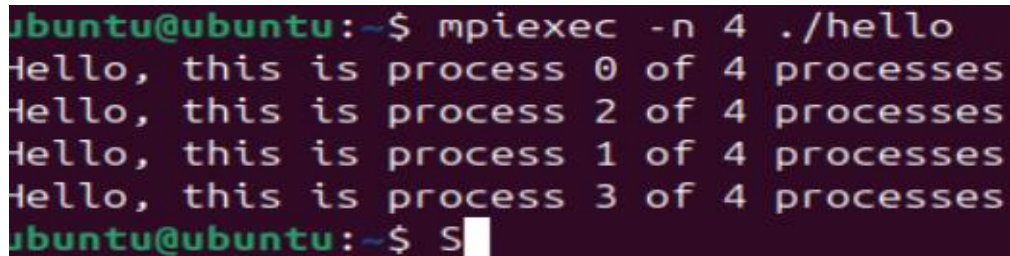
Mpi practice

1. Hello World with MPI

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

int main() {
    int size, rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello, this is process %d of %d processes \n", rank, size);
    MPI_Finalize();
    return 0;
}
```

Output :



```
ubuntu@ubuntu:~$ mpiexec -n 4 ./hello
Hello, this is process 0 of 4 processes
Hello, this is process 2 of 4 processes
Hello, this is process 1 of 4 processes
Hello, this is process 3 of 4 processes
ubuntu@ubuntu:~$ S
```

2. Sum of Array Elements (Distributed Summation)

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

int main(int argc, char** argv) {
    int rank;
    int size;
    int *array = NULL;
    int *sub_array = NULL;
    int local_sum = 0;
    int total_sum = 0;
    int n = 24;
    int chunk_size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    chunk_size = n / size;

    if (rank == 0) {
        array = (int*)malloc(n * sizeof(int));
        for (int i = 0; i < n; i++) {
            array[i] = i + 1;
        }
    }
}
```

```
sub_array = (int*)malloc(chunk_size * sizeof(int));

MPI_Scatter(array, chunk_size, MPI_INT, sub_array, chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

for (int i = 0; i < chunk_size; i++) {
    local_sum = local_sum + sub_array[i];
}

MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
    printf("Total sum = %d\n", total_sum);
    free(array);
}

free(sub_array);

MPI_Finalize();
return 0;
```

Output:

```
ubuntu@ubuntu:~$ mpiexec -n 4 ./sum
Total sum = 36
ubuntu@ubuntu:~$ mpicc sum.c -o sum
ubuntu@ubuntu:~$ mpiexec -n 4 ./sum
Total sum = 300
ubuntu@ubuntu:~$
```

3. Broadcast a Number:

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

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        number = 42;
    }

    MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process %d received number %d\n", rank, number);

    MPI_Finalize();
    return 0;
}
```

Output:

```
ubuntu@ubuntu:~$ mpicc Bcast.c -o Bcast
ubuntu@ubuntu:~$ mpiexec -n 4 ./Bcast
Process 0 received number 42
Process 1 received number 42
Process 2 received number 42
Process 3 received number 42
ubuntu@ubuntu:~$
```

4. Find Maximum Number

```
sum.c      Bcast.c

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

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    num = rank * 5;

    MPI_Reduce(&num, &max_number, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("The maximum number is %d\n", max_number);
    }

    MPI_Finalize();
    return 0;
}
```

Output:

```
ubuntu@ubuntu:~$ mpicc max.c -o max
ubuntu@ubuntu:~$ mpiexec -n 4 ./max
The maximum number is 15
ubuntu@ubuntu:~$
```

5. Ring Communication

sum.c	Bcast.c	max.c
<pre> include <stdio.h> nt main(int argc, char** argv) { int rank, size; int send_num, recv_num; int right, left; MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD, &size); send_num = rank; right = (rank + 1) % size; left = (rank - 1 + size) % size; // first send, then receive MPI_Send(&send_num, 1, MPI_INT, right, 0, MPI_COMM_WORLD); MPI_Recv(&recv_num, 1, MPI_INT, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); printf("Process %d received %d from process %d\n", rank, recv_num, left); MPI_Finalize(); return 0; </pre>		

Output:

```

ubuntu@ubuntu:~$ mpiexec -n 4 ./ring
Process 0 received 3 from process 3
Process 1 received 0 from process 0
Process 3 received 2 from process 2
Process 2 received 1 from process 1

```

6. Prefix Sum (Scan)

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

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    value = rank + 1;

    MPI_Scan(&value, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    printf("process %d: prefix sum = %d\n", rank, sum);

    MPI_Finalize();
    return 0;
}
```

Output:

```
ubuntu@ubuntu:~$ mpicc prefix.c -o prefix
ubuntu@ubuntu:~$ mpiexec -n 4 ./prefix
process 0: prefix sum = 1
process 1: prefix sum = 3
process 2: prefix sum = 6
process 3: prefix sum = 10
ubuntu@ubuntu:~$ S
```

7. Matrix Row Distribution

```
int main(int argc, char** argv) {
    int rank, size;
    int matrix[4][4];
    int row[4];
    int row_sum = 0;
    int all_sums[4];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        int value = 1;
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++) {
                matrix[i][j] = value;
                value++;
            }
        }
    }

    MPI_Scatter(matrix, 4, MPI_INT, row, 4, MPI_INT, 0, MPI_COMM_WORLD);

    for (int i = 0; i < 4; i++) {
        row_sum = row_sum + row[i];
    }

    MPI_Gather(&row_sum, 1, MPI_INT, all_sums, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Sums of each row:\n");
        for (int i = 0; i < 4; i++) {
            printf("Sum of row %d = %d\n", i, all_sums[i]);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Output:

```
Sums of each row:
Sum of row 0 = 10
Sum of row 1 = 26
Sum of row 2 = 42
Sum of row 3 = 58
ubuntu@ubuntu:~$
```

8. Barrier Synchronization with Execution Time Measurement:

```
int main(int argc, char** argv) {
    int rank, size;
    double start_time, end_time, elapsed_time;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Process %d: Before barrier \n", rank);

    start_time = MPI_Wtime();

    MPI_Barrier(MPI_COMM_WORLD);

    end_time = MPI_Wtime();
    elapsed_time = end_time - start_time;

    printf("Process %d: After barrier | Execution Time: %f seconds \n ", rank, elapsed_time);

    MPI_Finalize();
    return 0;
}
```

Output:

```
ubuntu@ubuntu:~$ mpiexec -n 3 ./barrier
Process 2: Before barrier
Process 1: Before barrier
Process 1: After barrier | Execution Time: 0.000112 seconds
Process 2: After barrier | Execution Time: 0.000149 seconds
Process 0: Before barrier
Process 0: After barrier | Execution Time: 0.000039 seconds
ubuntu@ubuntu:~$
```


9. Check Even or Odd Rank

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

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank % 2 == 0) {
        printf("Process %d is even\n", rank);
    } else {
        printf("Process %d is odd\n", rank);
    }

    MPI_Finalize();
    return 0;
}
```

Output:

```
ubuntu@ubuntu:~$ mpiexec -n 4 ./even
Process 0 is even
Process 1 is odd
Process 3 is odd
Process 2 is even
ubuntu@ubuntu:~$
```