

1. Introduction

This lab manual provides an overview of:

- The Linux environment and commonly used commands.
- Basics of OpenMP (for shared-memory parallel programming).
- Basics of MPI (for distributed-memory parallel programming).

It is intended for students beginning with **parallel and high-performance computing** in lab environments.

Part I – Linux Environment & Commands

1.1 Understanding the Linux Environment

Linux is an open-source operating system widely used in servers, HPC clusters, and development environments. It offers:

- **Multi-user capability**
- **Multitasking support**
- **Security and permissions**
- **Powerful command-line tools**

1.2 Essential Linux Commands

Category	Command	Description	Example
File/Directory	pwd	Show current directory	pwd
	ls	List files and folders	ls -l
	cd	Change directory	cd /home/user
	mkdir	Create directory	mkdir projects
	rm	Remove file	rm file.txt
	rmdir	Remove directory	rmdir testdir
	cp	Copy files	cp file1.txt file2.txt
	mv	Move/rename files	mv old.txt new.txt
File Viewing	cat	View file contents	cat file.txt
	less	Scroll through file	less bigfile.txt
	head	First 10 lines	head file.txt

	tail	Last 10 lines	tail -n 20 file.txt
Permissions	chmod	Change file permissions	chmod 755 script.sh
	chown	Change owner	chown user:user file.txt
Process Management	ps	View running processes	ps -u
	top	Monitor system usage	top
	kill	Kill process by PID	kill 1234
Searching	grep	Search in files	grep "error" log.txt
	find	Search for files	find / -name "*.c"
Networking	ping	Test connectivity	ping google.com
	scp	Secure copy between systems	scp file.txt user@host:/path
Package Management	apt / yum	Install packages	sudo apt install gcc

Lab Exercise 1 – Linux Basics

1. Create a directory named lab1.
2. Inside it, create a file hello.txt with some text.
3. Copy the file to backup.txt.
4. Change permissions so only you can read/write the file.
5. Use grep to search for a word inside it.
6. Delete the backup file.

Part II – OpenMP Basics

2.1 Introduction

OpenMP (Open Multi-Processing) is an **API for parallel programming** in C, C++, and Fortran for **shared-memory systems**.

Key Features:

- Uses **compiler directives** (`#pragma omp ...`) to parallelize code.
- Thread-based parallelism.
- Easy to add to existing sequential code.

2.2 Basic Structure

Example – Parallel Hello World:

```
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel
    {
        printf("Hello from thread %d\n", omp_get_thread_num());
    }
    return 0;
}
```

Compile and run:

```
gcc -fopenmp hello.c -o hello
./hello
```

Common OpenMP Directives

Directive	Purpose
<code>#pragma omp parallel</code>	Start a parallel region
<code>#pragma omp for</code>	Parallelize a loop
<code>#pragma omp sections</code>	Execute independent sections in parallel

#pragma omp single	Only one thread executes
#pragma omp critical	Protect critical sections
#pragma omp barrier	Synchronize threads

1. Write a OpenMP program to sort an array on n elements using both sequential and parallel mergesort(using Section). Record the difference in execution time.

```
#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>

#define SIZE 100000

// ----- MERGE FUNCTION -----

void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    i = 0; j = 0; k = left;
    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
    free(L);
    free(R);
}
```

```
}  
// ----- SERIAL MERGE SORT -----  
void serialMergeSort(int arr[], int left, int right) {  
    if (left < right) {  
        int mid = left + (right - left) / 2;  
        serialMergeSort(arr, left, mid);  
        serialMergeSort(arr, mid + 1, right);  
        merge(arr, left, mid, right);  
    }  
}  
// ----- PARALLEL MERGE SORT -----  
void parallelMergeSort(int arr[], int left, int right, int depth) {  
    if (left < right) {  
        int mid = left + (right - left) / 2;  
        if (depth <= 4) {  
            #pragma omp parallel sections  
            {  
                #pragma omp section  
                parallelMergeSort(arr, left, mid, depth + 1);  
                #pragma omp section  
                parallelMergeSort(arr, mid + 1, right, depth + 1);  
            }  
        }  
    }  
    else  
    {  
        // Switch to serial to avoid too many threads  
        serialMergeSort(arr, left, mid);  
        serialMergeSort(arr, mid + 1, right);  
    }  
}
```

```
    }
    merge(arr, left, mid, right);
}
}

// ----- MAIN FUNCTION -----

int main() {
    int *arr_serial = (int *)malloc(SIZE * sizeof(int));
    int *arr_parallel = (int *)malloc(SIZE * sizeof(int));

    // Initialize both arrays with the same random values
    for (int i = 0; i < SIZE; i++) {
        int val = rand() % 100000;
        arr_serial[i] = val;

        arr_parallel[i] = val;
    }

    // ----- SERIAL MERGE SORT -----
    clock_t start_serial = clock();
    serialMergeSort(arr_serial, 0, SIZE - 1);
    clock_t end_serial = clock(); // Fixed variable name
    double time_serial = (double)(end_serial - start_serial) / CLOCKS_PER_SEC;

    // ----- PARALLEL MERGE SORT -----
    clock_t start_parallel = clock();
    parallelMergeSort(arr_parallel, 0, SIZE - 1, 0);
    clock_t end_parallel = clock(); // Fixed variable name
    double time_parallel = (double)(end_parallel - start_parallel) / CLOCKS_PER_SEC;

    // ----- OUTPUT -----
    printf("Serial Merge Sort Time: %.6f seconds\n", time_serial);
```

```
printf("Parallel Merge Sort Time: %.6f seconds\n", time_parallel);  
// Optional: Verify correctness  
/*  
for (int i = 0; i < SIZE; i++) {  
    if (arr_serial[i] != arr_parallel[i]) {  
        printf("Mismatch at index %d: %d vs %d\n", i, arr_serial[i], arr_parallel[i]);  
        break;  
    }  
}  
free(arr_serial);  
free(arr_parallel);  
return 0;  
}
```

Case-1

```
#define SIZE 100000
```

```
gcc -fopenmp program1.c -o program1.out
```

```
./a.out
```

Serial Merge Sort Time: 0.015152 seconds

Parallel Merge Sort Time: 0.014014 seconds

Case-2

```
#define SIZE 10000000
```

```
cc program1.c
```

```
./a.out
```

Serial Merge Sort Time: 0.161685 seconds

Parallel Merge Sort Time: 0.159901 seconds

Case-3

```
#define SIZE 10000000
```

```
gcc -fopenmp program1.c -o program1.out
```

./a.out

Serial Merge Sort Time: 0.159916 seconds

Parallel Merge Sort Time: 0.157140 seconds

2. Write an OpenMP program that divides the Iterations into chunks containing 2 iterations, respectively (OMP_SCHEDULE=static,2). Its input should be the number of iterations, and its output should be which iterations of a parallelized for loop are executed by which thread.

For example, if there are two threads and four iterations, the output might be the following:

Thread 0 : Iterations 0 — 1

Thread 1 : Iterations 2 – 3

Code	Comment
#include<stdio.h>	
#include<omp.h>	
int main()	
{	
int n = 16,thread,i;	
printf("\n Enter the number of tasks");	
scanf("%d",&n);	
printf("\n n Enter the number of threads");	
scanf("%d",&thread);	
omp_set_num_threads(thread);	
printf("\n ----- \n");	
#pragma omp parallel for schedule(static, 2)	
for (i = 0; i< n; i++)	
{	
printf("Thread %d executes iteration %d %d\n",n,	
omp_get_thread_num(), i);	
}	
return 0;	
}	

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int n = 16,thread,i;
    printf("\n Enter the number of tasks");
    scanf("%d",&n);
    printf("\n n Enter the number of threads");
    scanf("%d",&thread);
    omp_set_num_threads(thread);
    printf("\n -----\\n");
    #pragma omp parallel for schedule(static, 2)
    for (i = 0; i< n; i++)
    {
        printf("Thread %d executes iteration %d %d\\n",n, omp_get_thread_num(), i);
    }
    return 0;
}
```

Case-1

gcc program2.c -fopenmp

./a.out

Enter the number of tasks12

n Enter the number of threads6

-----\\n
Thread 12 executes iteration 0 0
Thread 12 executes iteration 0 1
Thread 12 executes iteration 4 8
Thread 12 executes iteration 4 9
Thread 12 executes iteration 1 2
Thread 12 executes iteration 1 3
Thread 12 executes iteration 5 10
Thread 12 executes iteration 5 11
Thread 12 executes iteration 3 6
Thread 12 executes iteration 3 7
Thread 12 executes iteration 2 4
Thread 12 executes iteration 2 5

Case-1

./a.out

Enter the number of tasks 24

n Enter the number of threads 12

-----\\n
Thread 24 executes iteration 6 12

Thread 24 executes iteration 6 13
Thread 24 executes iteration 2 4
Thread 24 executes iteration 2 5
Thread 24 executes iteration 11 22
Thread 24 executes iteration 11 23
Thread 24 executes iteration 5 10
Thread 24 executes iteration 5 11
Thread 24 executes iteration 7 14
Thread 24 executes iteration 7 15
Thread 24 executes iteration 8 16
Thread 24 executes iteration 8 17
Thread 24 executes iteration 9 18
Thread 24 executes iteration 9 19
Thread 24 executes iteration 4 8
Thread 24 executes iteration 4 9
Thread 24 executes iteration 1 2
Thread 24 executes iteration 1 3
Thread 24 executes iteration 0 0
Thread 24 executes iteration 0 1
Thread 24 executes iteration 3 6
Thread 24 executes iteration 3 7
Thread 24 executes iteration 10 20
Thread 24 executes iteration 10 21

3. Write a OpenMP program to calculate n Fibonacci numbers using tasks.

```
#include <stdio.h>
#include <omp.h>
#include <time.h>
// Serial Fibonacci calculation function
int ser_fib(long int n)
{
    if (n < 2) return n;
    return ser_fib(n - 1) + ser_fib(n - 2);
}

// Parallel Fibonacci calculation function
int fib(long int n)
{
    if (n < 2) return n;

    int x, y;

    #pragma omp task shared(x)
    x = fib(n - 1);

    #pragma omp task shared(y)
    y = fib(n - 2);

    #pragma omp taskwait
    return x + y;
}

int main()
{
    long int n = 10;
    int result;
    clock_t start, end;
    double cpu_time;

    printf("Enter the value of n: ");
    scanf("%ld", &n);

    // Parallel Fibonacci
    start = clock();
    #pragma omp parallel
    {
        #pragma omp single
        result = fib(n);
    }
}
```

```
end = clock();
cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("\nParallel Fibonacci(%ld) = %d", n, result);
printf("\nTime used in parallel mode: %f seconds\n", cpu_time);

// Serial Fibonacci
start = clock();
result = ser_fib(n);
end = clock();
cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("\nSerial Fibonacci(%ld) = %d", n, result);
printf("\nTime used in serial mode: %f seconds\n", cpu_time);
return 0;
}
```

Case-2

cc program3_fib.c

./a.out

Enter the value of n: 10

Parallel Fibonacci(10) = 55

Time used in parallel mode: 0.000001 seconds

Serial Fibonacci(10) = 55

Time used in serial mode: 0.000001 seconds

Case-2

./a.out

Enter the value of n: 32

Parallel Fibonacci(32) = 2178309

Time used in parallel mode: 0.016277 seconds

Serial Fibonacci(32) = 2178309

Time used in serial mode: 0.009735 seconds

4. Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times.

```
#include <stdio.h>
#include <omp.h>
#include <time.h>
int is_prime(int n)
{
    if (n < 2)
        return 0;
    for (int j = 2; j * j <= n; j++)
    {
        if (n % j == 0)
            return 0;
    }
    return 1;
}
int main()
{
    long n = 10000000,i;
    clock_t start, end;
    double cpu_time;
    printf("\nThe range of numbers is 1 to %ld\n", n);
    printf("\n-----\n");

    // Serial Execution
    start = clock();
    for (i = 0; i < n; i++)
    {
        is_prime(i);
    }
    end = clock();
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time to compute prime numbers serially: %f seconds\n", cpu_time);

    // Parallel Execution
    start = clock();
    #pragma omp parallel for
    for ( i = 0; i <n; i++)
    {
        is_prime(i);
    }
    end = clock();
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time to compute prime numbers in parallel: %f seconds\n", cpu_time);
```

```
    return 0;  
}
```

Output1

```
cc program4_primenumber.c  
gcc program4_primenumber.c -fopenmp  
./a.out  
The range of numbers is 1 to 10000000
```

Time to compute prime numbers serially: 3.851528 seconds
Time to compute prime numbers in parallel: 6.496520 seconds

Output2

```
cc program4_primenumber.c  
gcc program4_primenumber.c -fopenmp  
./a.out  
The range of numbers is 1 to 1000000
```

Time to compute prime numbers serially: 0.156123 seconds
Time to compute prime numbers in parallel: 0.292745 seconds

Output3

```
cc program4_primenumber.c  
gcc program4_primenumber.c -fopenmp  
  
./a.out  
The range of numbers is 1 to 100000
```

Time to compute prime numbers serially: 0.009664 seconds
Time to compute prime numbers in parallel: 0.014261 seconds

Part III – MPI Basics

3.1 Introduction

MPI (**M**essage **P**assing **I**nterface) is a standard for parallel programming on **distributed-memory systems** like clusters.

Key Features:

- Multiple processes (not threads).
- Explicit communication using send and receive.
- Portable across platforms.

3.2 Basic Structure

Example – Parallel Hello World:

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

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

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Process ID
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Total processes

    printf("Hello from process %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}

Compile and run:
mpicc hello_mpi.c -o hello_mpi
mpirun -np 4 ./hello_mpi
```

3.3 Common MPI Functions

Function	Purpose
MPI_Init	Initialize MPI
MPI_Finalize	End MPI program
MPI_Comm_rank	Get process ID
MPI_Comm_size	Get number of processes
MPI_Send	Send data
MPI_Recv	Receive data
MPI_Bcast	Broadcast data to all processes
MPI_Reduce	Combine values from all processes

Sample hello program in mpi

Library to be installed on ubuntu to run MPI programs

sudo apt install openmpi-bin openmpi-common libopenmpi-dev

Creating a simple program

gedit hello.c

Code:

```
//hello.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);
    printf("Hello from processor %d\n", rank);
}
```

```
MPI_Finalize();  
return 0;  
}
```

Execution

mpicc hello.c -o hello

To observe Results

Case-1:

mpirun ./hello

Output

```
Hello from processor 5  
Hello from processor 2  
Hello from processor 4  
Hello from processor 3  
Hello from processor 6  
Hello from processor 0  
Hello from processor 1  
Hello from processor 7
```

Case -2:

mpirun -np 2 ./hello

```
Hello from processor 1  
Hello from processor 0
```

Case -3:

mpirun -np 4 ./hello

```
Hello from processor 3  
Hello from processor 0  
Hello from processor 1  
Hello from processor 2
```

Case -4:

mpirun -np 8./hello

Hello from processor 2
Hello from processor 1
Hello from processor 6
Hello from processor 7
Hello from processor 5
Hello from processor 4
Hello from processor 0
Hello from processor 3

Case -5: mpirun -np 6./hello

Hello from processor 0
Hello from processor 1
Hello from processor 3
Hello from processor 4
Hello from processor 5
Hello from processor 2

Case -6: mpirun -np 16./hello

There are not enough slots available in the system to satisfy the 16 slots that were requested by the application:

./hello

Either request fewer slots for your application, or make more slots available for use.

A "slot" is the Open MPI term for an allocatable unit where we can launch a process. The number of slots available are defined by the environment in which Open MPI processes are run:

1. Hostfile, via "slots=N" clauses (N defaults to number of processor cores if not provided)
2. The --host command line parameter, via a ":N" suffix on the

hostname (N defaults to 1 if not provided)

3. Resource manager (e.g., SLURM, PBS/Torque, LSF, etc.)

4. If none of a hostfile, the --host command line parameter, or an

RM is present, Open MPI defaults to the number of processor cores

In all the above cases, if you want Open MPI to default to the number of hardware threads instead of the number of processor cores, use the --use-hwthread-cpus option.

Alternatively, you can use the --oversubscribe option to ignore the number of available slots when deciding the number of processes to launch.

Type `lscpu` to check no. of cores, thread & socket availability.

Program 5:

Write a MPI Program to demonstration of MPI_Send and MPI_Recv.

STEP1: Open terminal(ctrl+alt+t) short cut key to open terminal

Type

gedit sr.c

type code as in below

```
// sr.c

#include <mpi.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char** argv)
{
    int rank, size;
    MPI_Status status;

    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Get current process rank and total number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Process 0 sends a message to all other processes
    if (rank == 0)
    {
        for (int i = 1; i < size; i++)
        {
            char message[100];
            sprintf(message, "Hello from process 0 to process %d", i);
            MPI_Send(message, strlen(message) + 1, MPI_CHAR, i, 0,
                MPI_COMM_WORLD);
            printf("Process 0 sent message to process %d\n", i);
        }
    }
    else
    {
        // All other processes receive the message from process 0
    }
}
```

```
char recv_buf[100];
MPI_Recv(recv_buf, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
&status);
printf("Process %d received message: %s\n", rank, recv_buf);
}

// Finalize MPI
MPI_Finalize();
return 0;
}
```

Now For Compilation

mpicc sr.c -o sr

For execution

Case-1)

Output

mpirun -oversubscribe --np 6 ./sr

Process 0 sent message to process 1

Process 0 sent message to process 2

Process 0 sent message to process 3

Process 0 sent message to process 4

Process 0 sent message to process 5

Process 4 received message: Hello from process 0 to process 4

Process 5 received message: Hello from process 0 to process 5

Process 2 received message: Hello from process 0 to process 2

Process 3 received message: Hello from process 0 to process 3

Process 1 received message: Hello from process 0 to process 1

Case-2:

Output

mpirun -oversubscribe --np 16 ./sr

Process 0 sent message to process 1

Process 0 sent message to process 2

Process 0 sent message to process 3
Process 0 sent message to process 4
Process 0 sent message to process 5
Process 0 sent message to process 6
Process 0 sent message to process 7
Process 0 sent message to process 8
Process 2 received message: Hello from process 0 to process 2
Process 7 received message: Hello from process 0 to process 7
Process 3 received message: Hello from process 0 to process 3
Process 0 sent message to process 9
Process 0 sent message to process 10
Process 0 sent message to process 11
Process 0 sent message to process 12
Process 0 sent message to process 13
Process 0 sent message to process 14
Process 0 sent message to process 15
Process 11 received message: Hello from process 0 to process 11
Process 12 received message: Hello from process 0 to process 12
Process 14 received message: Hello from process 0 to process 14
Process 15 received message: Hello from process 0 to process 15
Process 4 received message: Hello from process 0 to process 4
Process 5 received message: Hello from process 0 to process 5
Process 6 received message: Hello from process 0 to process 6
Process 9 received message: Hello from process 0 to process 9
Process 8 received message: Hello from process 0 to process 8
Process 10 received message: Hello from process 0 to process 10
Process 13 received message: Hello from process 0 to process 13
Process 1 received message: Hello from process 0 to process 1

Program 6:

Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence

STEP1: Open terminal

Type

gedit deadlock.c

type code as in below

```
// deadlock.c

#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    int rank, size;
    int send_data, recv_data;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)
    {
        send_data = 100;
        // blocking send&recieved
        MPI_Send(&send_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&recv_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
        printf("Process 0 received %d\n", recv_data);
    }
    else if (rank == 1)
    {
        send_data = 200;
        MPI_Send(&send_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); // blocking send
        MPI_Recv(&recv_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received %d\n", recv_data);
    }
    MPI_Finalize();
    return 0;
}
```

```
}
```

Now For Compilation

```
mpicc deadlock.c -o deadlock
```

For execution

Case-1):

```
mpirun --np 2 ./deadlock
```

```
Process 0 received 200
```

```
Process 1 received 100
```

Case-2):

```
mpirun --np 8 ./ex_6
```

```
Process 0 received 200
```

```
Process 1 received 100
```

```
#include <mpi.h>

#include <stdio.h>

#include <math.h>


// Function to integrate
double f(double x) {
    return x * x; // Example: f(x) = x^2
}


// Trapezoidal rule implementation for local process
double trapezoidal_rule(double a, double b, int n, double h) {
    double result = (f(a) + f(b)) / 2.0;
    for (int i = 1; i < n; i++) {
        double x = a + i * h;
        result += f(x);
    }
    return result * h;
}


int main(int argc, char** argv) {
    int rank, size, n = 1000;
    double a = 0.0, b = 10.0; // Interval [a, b]
    double h, local_a, local_b, local_integral, total_integral;
    int local_n;

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

```
h = (b - a) / n;      // Step size
local_n = n / size;   // Number of trapezoids per process

local_a = a + rank * local_n * h;
local_b = local_a + local_n * h;

// Each process calculates its local integral
local_integral = trapezoidal_rule(local_a, local_b, local_n, h);

// Reduce all local integrals to process 0
MPI_Reduce(&local_integral, &total_integral, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

if (rank == 0) {
    printf("With n = %d trapezoids, integral from %f to %f = %f\n", n, a, b, total_integral);
}

MPI_Finalize();
return 0;
}
```

Program 7:

Write a MPI Program to demonstration of Broadcast operation.

STEP1: Open terminal

Type

gedit broadcast.c

type code as in below

```
// broadcast.c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    int rank, size;
    int number;
    // Initialize MPI environment
    MPI_Init(&argc, &argv);
    // Get current process rank and total number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0)
    {
        number = 42; // The value to be broadcasted
        printf("Process 0 is broadcasting number %d\n", number);
    }

    // Broadcast the value of `number` from process 0 to all others
    MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // All processes print the received number
    printf("Process %d received number %d\n", rank, number);

    // Finalize MPI
    MPI_Finalize();
    return 0;
}
```

Now For Compilation

mpicc broadcast.c -o broadcast

For execution

mpirun -oversubscribe --np 4 ./broadcast

Results:

Case-1)

```
mpirun ./ex_7
```

Process 0 is broadcasting number 42

Process 0 received number 42

Process 3 received number 42

Process 5 received number 42

Process 6 received number 42

Process 7 received number 42

Process 1 received number 42

Process 2 received number 42

Process 4 received number 42

Case:2)

```
mpirun --np 4 ./ex_7
```

Process 0 is broadcasting number 42

Process 0 received number 42

Process 1 received number 42

Process 2 received number 42

Process 3 received number 42

Program 8:

Write a MPI Program demonstration of MPI_Scatter and MPI_Gather

STEP1: Open terminal

Type

gedit sg.c

type code as in below

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    int rank, size;
    int data[100];    // Only used by root
    int recv_value;    // Value received by each process
    int gathered[100]; // Only used by root

    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Get rank and size
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Only rank 0 initializes data
    if (rank == 0)
    {
        for (int i = 0; i < size; i++)
        {
            data[i] = i * 10;
        }
        printf("Process 0 initialized data: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

    // SCATTER: One value from `data` array is sent to each process
    MPI_Scatter(data, 1, MPI_INT, &recv_value, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d received value %d from Scatter\n", rank, recv_value);
```

```
// Each process modifies its value
recv_value += rank;

// GATHER: Each process sends modified value back to root
MPI_Gather(&recv_value, 1, MPI_INT, gathered, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Only rank 0 displays gathered results
if (rank == 0)
{
    printf("Process 0 gathered data: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", gathered[i]);
    }
    printf("\n");
}

// Finalize
MPI_Finalize();
return 0;
}
```

Now For Compilation

mpicc sg.c -o sg

For execution

Case-1)

mpirun -oversubscribe --np 4 ./sg

Process 0 initialized data: 0 10 20 30

Process 0 received value 0 from Scatter

Process 1 received value 10 from Scatter

Process 3 received value 30 from Scatter

Process 0 gathered data: 0 11 22 33

Process 2 received value 20 from Scatter

Case-2)

mpirun --np 7 ./ex_8

Process 0 initialized data: 0 10 20 30 40 50 60

Process 1 received value 10 from Scatter
Process 2 received value 20 from Scatter
Process 3 received value 30 from Scatter
Process 4 received value 40 from Scatter
Process 5 received value 50 from Scatter
Process 0 received value 0 from Scatter
Process 0 gathered data: 0 11 22 33 44 55 66
Process 6 received value 60 from Scatter

Case-3)

mpirun ./ex_8
Process 0 initialized data: 0 10 20 30 40 50 60 70
Process 0 received value 0 from Scatter
Process 1 received value 10 from Scatter
Process 5 received value 50 from Scatter
Process 2 received value 20 from Scatter
Process 4 received value 40 from Scatter
Process 3 received value 30 from Scatter
Process 7 received value 70 from Scatter
Process 6 received value 60 from Scatter
Process 0 gathered data: 0 11 22 33 44 55 66 77

Program 9:

Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

STEP1: Open terminal

Type

gedit pg9.c

type code as in below

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    int rank, size;
    int value;
    int sum_result, prod_result, max_result, min_result;
    int all_sum, all_prod, all_max, all_min;

    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Get rank and size
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Each process sets its value
    value = rank + 1; // Example: rank 0 = 1, rank 1 = 2, etc.
    printf("Process %d has value %d\n", rank, value);

    // ----- MPI_Reduce (result only at root, rank 0) -----
    MPI_Reduce(&value, &sum_result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &prod_result, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &max_result, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &min_result, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("\n--- Results using MPI_Reduce (only on root) ---\n");
        printf("Sum = %d\n", sum_result);
    }
}
```

```
    printf("Prod = %d\n", prod_result);
    printf("Max  = %d\n", max_result);
    printf("Min  = %d\n", min_result);
}

// ----- MPI_Allreduce (result sent to all processes) -----
MPI_Allreduce(&value, &all_sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(&value, &all_prod, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
MPI_Allreduce(&value, &all_max, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
MPI_Allreduce(&value, &all_min, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

printf("Process %d - AllReduce: Sum=%d Prod=%d Max=%d Min=%d\n",
       rank, all_sum, all_prod, all_max, all_min);

// Finalize
MPI_Finalize();
return 0;
}
```

Now For Compilation

mpicc pg9.c -o pg9

For execution

Case-1)

mpirun ./ex_9

Process 4 has value 5

Process 7 has value 8

Process 2 has value 3

Process 3 has value 4

Process 5 has value 6

Process 1 has value 2

Process 6 has value 7

Process 0 has value 1

--- Results using MPI_Reduce (only on root) ---

Sum = 36

Prod = 40320

Max = 8

Min = 1

Process 2 - AllReduce: Sum=36 Prod=40320 Max=8 Min=1

Process 3 - AllReduce: Sum=36 Prod=40320 Max=8 Min=1

Process 1 - AllReduce: Sum=36 Prod=40320 Max=8 Min=1

Process 4 - AllReduce: Sum=36 Prod=40320 Max=8 Min=1

Process 0 - AllReduce: Sum=36 Prod=40320 Max=8 Min=1

Process 6 - AllReduce: Sum=36 Prod=40320 Max=8 Min=1

Process 5 - AllReduce: Sum=36 Prod=40320 Max=8 Min=1

Process 7 - AllReduce: Sum=36 Prod=40320 Max=8 Min=1

Case-2)

mpirun --np 2 ./pg9

mpirun --np 2 ./ex_9

Process 0 has value 1

Process 1 has value 2

--- Results using MPI_Reduce (only on root) ---

Sum = 3

Prod = 2

Max = 2

Min = 1

Process 1 - AllReduce: Sum=3 Prod=2 Max=2 Min=1

Process 0 - AllReduce: Sum=3 Prod=2 Max=2 Min=1

Case-3)

mpirun --np 6 ./ex_9

Process 3 has value 4

Process 2 has value 3

Process 5 has value 6

Process 1 has value 2

Process 4 has value 5

Process 0 has value 1

--- Results using MPI_Reduce (only on root) ---

Sum = 21

Prod = 720

Max = 6

Min = 1

Process 0 - AllReduce: Sum=21 Prod=720 Max=6 Min=1

Process 4 - AllReduce: Sum=21 Prod=720 Max=6 Min=1

Process 5 - AllReduce: Sum=21 Prod=720 Max=6 Min=1

Process 3 - AllReduce: Sum=21 Prod=720 Max=6 Min=1

Process 1 - AllReduce: Sum=21 Prod=720 Max=6 Min=1

Process 2 - AllReduce: Sum=21 Prod=720 Max=6 Min=1