**Mpi\_Sum\_avg**

#include <stdio.h>

#include<stdlib.h>

#include<mpi.h>

#define ARRAY\_SIZE 1000

int main(int argc,char \*argv[])

{

int rank,size;

int i;

int local\_sum=0;

double local\_avg=0.0;

int total\_sum=0;

double total\_avg=0.0;

int local\_sums[size];

double total\_avgs[size];

int numbers[ARRAY\_SIZE];

MPI INit(&argc,&argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);

if(rank==0)

{

for(i=0;i<ARRAY\_SIZE;i++)

{

numbers[i]=rand()%d 100;

}

MPI\_Scatter(numbers,ARRAY\_SIZE/size,MPI\_INT,numbers,ARRAY\_SIZE/size,MPI\_INT,0,MPI\_COMM\_WORLD);

}

for(i=0;i<ARRAY\_SIZE/size;i++)

{

local\_sum+=numbers[i];

}

local\_avg=(double)local\_sum/(ARRAY\_SIZE/size);

MPI\_Gather(&local\_sum,0,MPI\_INT,local\_sums,0,MPI\_INT,MPI\_COMM\_WORLD)

MPI\_Gather(&local\_avg,0,MPI\_DOUBLE,local\_avgs,0,MPI\_DOUBLE,MPI\_COMM\_WORLD)

if(rank==0)

for(i=0;i<size;i++)

{

total\_sum+=local\_sums[i];

total\_avg+=local\_avgs[i];

}

total\_avg/=size;

printf("total sum of 1000 random numbers:%d\n",total\_sum);

printf("total avg of 1000 random numbers:%.2f\n",total\_avg);

}

MPI\_finalized();

return 0;

**mpi\_min**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#define ARRAY\_SIZE 1000

int main(int argc, char\*\* argv) {

int rank, size;

int numbers[ARRAY\_SIZE];

int min\_number = 0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

// Generate random numbers for the root process

for (int i = 0; i < ARRAY\_SIZE; i++) {

numbers[i] = rand() % 1000; // Generate random numbers between 0 and 999

}

}

// Scatter the array to all processes

MPI\_Scatter(numbers, ARRAY\_SIZE/size, MPI\_INT, numbers, ARRAY\_SIZE/size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Find the minimum number in the local array

int local\_min = numbers[0];

for (int i = 1; i < ARRAY\_SIZE/size; i++) {

if (numbers[i] < local\_min) {

local\_min = numbers[i];

}

}

// Reduce all the local minimums to find the global minimum

MPI\_Reduce(&local\_min, &min\_number, 1, MPI\_INT, MPI\_MIN, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

printf("The minimum number is: %d\n", min\_number);

}

MPI\_Finalize();

return 0;

}

**Mpi\_max**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#define SIZE 1000

int main(int argc, char\*\* argv) {

int rank, size;

int numbers[SIZE];

int max\_number = 0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

// Generate random numbers for the root process

for (int i = 0; i < SIZE; i++) {

numbers[i] = rand() % 1000; // Generate numbers between 0 and 999

}

}

// Scatter the numbers to all processes

MPI\_Bcast(numbers, SIZE, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Find the local maximum

for (int i = 0; i < SIZE; i++) {

if (numbers[i] > max\_number) {

max\_number = numbers[i];

}

}

// Reduce to find the global maximum

int global\_max\_number;

MPI\_Reduce(&max\_number, &global\_max\_number, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

printf("The maximum number is: %d\n", global\_max\_number);

}

MPI\_Finalize();

return 0;

}

**Sstf**

#include<stdlib.h>

#include<limits.h>

int SSTF(int \*RQ, int n, int head) {

int min, diff, index, totalHeadMoment=0, count=0, i;

printf("\nOrder of serving requests: ");

while(count != n) {

min = INT\_MAX;

for(i=0; i<n; i++) {

diff = abs(head - RQ[i]);

if(diff < min) {

min = diff;

index = i;

}

}

printf("%d ", RQ[index]);

totalHeadMoment += min;

head = RQ[index];

RQ[index] = INT\_MAX;

count++;

}

return totalHeadMoment;

}

int main() {

int RQ[100], i, n, totalHeadMoment, head, totalSize;

printf("\nEnter the max range of the disk: ");

scanf("%d", &totalSize);

printf("\nEnter the size of queue request: ");

scanf("%d", &n);

printf("\nEnter the queue of disk positions to be read:\n");

for(i=0; i<n; i++)

scanf("%d",&RQ[i]);

printf("\nEnter initial head position: ");

scanf("%d", &head);

totalHeadMoment = SSTF(RQ, n, head);

}

**Conti**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_BLOCKS 1000

int bit\_vector[MAX\_BLOCKS];

void initialize(int n) {

srand(time(NULL)); // Seed the random number generator with the current time

for (int i = 0; i < n; i++) {

if (rand() % 2 == 0) {

bit\_vector[i] = 1; // Mark block as allocated

} else {

bit\_vector[i] = 0; // Mark block as free

}

}

}

void show\_bit\_vector(int n) {

printf("Block Number\tStatus\n");

for (int i = 0; i < n; i++) {

printf("%d\t\t", i);

if (bit\_vector[i] == 1) {

printf("Allocated\n");

} else {

printf("Free\n");

}

}

}

void create\_new\_file(int n) {

int start\_block, num\_blocks;

printf("Enter the starting block number: ");

scanf("%d", &start\_block);

printf("Enter the number of blocks needed: ");

scanf("%d", &num\_blocks);

int i;

for (i = start\_block; i < start\_block + num\_blocks; i++) {

if (i >= n || bit\_vector[i] == 1) {

printf("Error: Cannot allocate file in the specified blocks.\n");

break;

}

}

if (i == start\_block + num\_blocks) {

printf("File created successfully.\n");

for (i = start\_block; i < start\_block + num\_blocks; i++) {

bit\_vector[i] = 1; // Mark blocks as allocated

}

}

}

void show\_directory(int n) {

printf("File Name\tStarting Block\tNumber of Blocks\n");

// TODO: Implement directory functionality here

}

int main() {

int n, choice;

printf("Enter the number of blocks in the disk: ");

scanf("%d", &n);

initialize(n);

do {

printf("\nMenu:\n");

printf("1. Show Bit Vector\n");

printf("2. Create New File\n");

printf("3. Show Directory\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

show\_bit\_vector(n);

break;

case 2:

create\_new\_file(n);

break;

case 3:

show\_directory(n);

break;

case 4:

printf("Exiting...\n");

break;

default:

printf("Error: Invalid choice.\n");

break;

}

} while (choice != 4);

return 0;

}