**CSE4001-Parallel and distributed computing**

**Name:** OG RAGAVI

**Reg No:** 20BCE1988

**Lab Ex:** 9

**Date:** 27.09.2022

**Important commands:**

* **MPI\_Init (NULL, NULL)** –Initializes the MPI program.
* **MPI\_Comm\_size(MPI\_COMM\_WORLD,&no of processors)** –tells how many processors are there in communication.
* **int rank** – indicates the processor id according to its priority.
* **int MPI\_Send(void \*data\_to\_send, ​int send\_count, ​ MPI\_Datatype send\_type, ​int destination\_ID, ​ int tag, MPI\_Comm comm)**- used to send the data to the destination processor.Tag value depends on the number of different processes allocated to the system.If we have only one function, we’ll use tag=0.
* **int MPI\_Recv(void \*received\_data, ​ int receive\_count, ​MPI\_Datatype receive\_type,int sender\_ID, ​ int tag, ​ MPI\_Comm comm, ​MPI\_Status \*status) –**used to receive data from a particular processor.Status maintains the success result.

Q1) Program to sort an array of 100 numbers.

**Algorithm:**

* Initialize the MPI program to run for multiple processors.
* Using rand() function , take input of 100 random numbers.
* Divide the program to np processors= no of elements in the array/np.
* Initially the master program gets into action and sorts the first 25 elements(in case of 100 numbers , as 100/4 processors= 25).
* Then it passes the index one by one (incrementing by 25) to the processor with pid 1, 2 and so on till np.
* Each slave computes the sorted array of their particular subpart. For eg)Pid=1 sorts the array from index 26 to 50 and so on.
* These partially sorted arrays are sent to the master processor , which are finally merged and sorted once again to display the output>

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <mpi.h>

//sort for 100 numbers

void merge(int \*, int \*, int, int, int);

void mergeSort(int \*, int \*, int, int);

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

/\*\*\*\*\*\*\*\*\*\* Create and populate the array \*\*\*\*\*\*\*\*\*\*/

int n = atoi(argv[1]);

int \*original\_array = malloc(n \* sizeof(int));

int c;

srand(time(NULL));

printf("The initial unsorted array is: ");

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

original\_array[c] = rand() % n;

printf("%d ", original\_array[c]);

}

printf("\n");

printf("\n");

/\*\*\*\*\*\*\*\*\*\* Initialize MPI \*\*\*\*\*\*\*\*\*\*/

int world\_rank;//priority given to the processor

int world\_size;

MPI\_Init(&argc, &argv);

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

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

/\*\*\*\*\*\*\*\*\*\* Divide the array in equal-sized chunks \*\*\*\*\*\*\*\*\*\*/

int size = n/world\_size;

/\*\*\*\*\*\*\*\*\*\* Send each subarray to each process \*\*\*\*\*\*\*\*\*\*/

int \*sub\_array = malloc(size \* sizeof(int));

MPI\_Scatter(original\_array, size, MPI\_INT, sub\_array, size, MPI\_INT, 0, MPI\_COMM\_WORLD);

/\*\*\*\*\*\*\*\*\*\* Perform the mergesort on each process \*\*\*\*\*\*\*\*\*\*/

int \*tmp\_array = malloc(size \* sizeof(int));

mergeSort(sub\_array, tmp\_array, 0, (size - 1));

/\*\*\*\*\*\*\*\*\*\* Gather the sorted subarrays into one \*\*\*\*\*\*\*\*\*\*/

int \*sorted = NULL;

if(world\_rank == 0) {

sorted = malloc(n \* sizeof(int));

}

MPI\_Gather(sub\_array, size, MPI\_INT, sorted, size, MPI\_INT, 0, MPI\_COMM\_WORLD);

/\*\*\*\*\*\*\*\*\*\* Make the final mergeSort call \*\*\*\*\*\*\*\*\*\*/

if(world\_rank == 0) {

int \*other\_array = malloc(n \* sizeof(int));

mergeSort(sorted, other\_array, 0, (n - 1));

/\*\*\*\*\*\*\*\*\*\* Display the sorted array \*\*\*\*\*\*\*\*\*\*/

printf("The final sorted array is: ");

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

printf("%d ", sorted[c]);

}

printf("\n");

printf("\n");

/\*\*\*\*\*\*\*\*\*\* Clean up root \*\*\*\*\*\*\*\*\*\*/

free(sorted);

free(other\_array);

}

/\*\*\*\*\*\*\*\*\*\* Clean up rest \*\*\*\*\*\*\*\*\*\*/

free(original\_array);

free(sub\_array);

free(tmp\_array);

/\*\*\*\*\*\*\*\*\*\* Finalize MPI \*\*\*\*\*\*\*\*\*\*/

MPI\_Barrier(MPI\_COMM\_WORLD);

MPI\_Finalize();

}

/\*\*\*\*\*\*\*\*\*\* Merge Function \*\*\*\*\*\*\*\*\*\*/

void merge(int \*a, int \*b, int l, int m, int r) {

int h, i, j, k;

h = l;

i = l;

j = m + 1;

while((h <= m) && (j <= r)) {

if(a[h] <= a[j]) {

b[i] = a[h];

h++;

}

else {

b[i] = a[j];

j++;

}

i++;

}

if(m < h) {

for(k = j; k <= r; k++) {

b[i] = a[k];

i++;

}

}

else {

for(k = h; k <= m; k++) {

b[i] = a[k];

i++;

}

}

for(k = l; k <= r; k++) {

a[k] = b[k];

}

}

/\*\*\*\*\*\*\*\*\*\* Recursive Merge Function \*\*\*\*\*\*\*\*\*\*/

void mergeSort(int \*a, int \*b, int l, int r) {

int m;

if(l < r) {

m = (l + r)/2;

mergeSort(a, b, l, m);

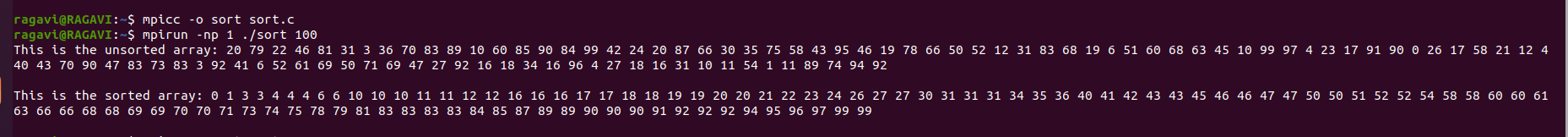
mergeSort(a, b, (m + 1), r);

merge(a, b, l, m, r);

}

}

**Output:**



Q2) Program to perform matrix multiplication:

**Code:**

#include <mpi.h>

#include <stdio.h>

#include<stdlib.h>

#include<math.h>

#define SIZE 8 /\* Size of matrices \*/

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

void fill\_matrix(int m[SIZE][SIZE])

{

static int n=0;

int i, j;

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

for (j=0; j<SIZE; j++)

//m[i][j] = n++;

m[i][j]=rand()%50;

}

void print\_matrix(int m[SIZE][SIZE])

{

int i, j = 0;

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

printf("\n\t| ");

for (j=0; j<SIZE; j++)

printf("%2d ", m[i][j]);

printf("|");

}

}

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

{

int myrank, P, from, to, i, j, k;

int tag = 666; /\* any value will do \*/

MPI\_Status status;

MPI\_Init (&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank); /\* who am i \*/

MPI\_Comm\_size(MPI\_COMM\_WORLD, &P); /\* number of processors \*/

/\* Just to use the simple variants of MPI\_Gather and MPI\_Scatter we \*/

/\* impose that SIZE is divisible by P. By using the vector versions, \*/

/\* (MPI\_Gatherv and MPI\_Scatterv) it is easy to drop this restriction. \*/

if (SIZE%P!=0) {

if (myrank==0) printf("Matrix size not divisible by number of processors\n");

MPI\_Finalize();

exit(-1);

}

from = myrank \* SIZE/P;

to = (myrank+1) \* SIZE/P;

/\* Process 0 fills the input matrices and broadcasts them to the rest \*/

/\* (actually, only the relevant stripe of A is sent to each process) \*/

if (myrank==0) {

fill\_matrix(A);

fill\_matrix(B);

}

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

MPI\_Scatter (A, SIZE\*SIZE/P, MPI\_INT, A[from], SIZE\*SIZE/P, MPI\_INT, 0, MPI\_COMM\_WORLD);

printf("Result of matrix multiplication is:");

for (i=from; i<to; i++)

for (j=0; j<SIZE; j++) {

C[i][j]=0;

for (k=0; k<SIZE; k++)

C[i][j] += A[i][k]\*B[k][j];

}

MPI\_Gather (C[from], SIZE\*SIZE/P, MPI\_INT, C, SIZE\*SIZE/P, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (myrank==0) {

printf("\n\n");

print\_matrix(A);

printf("\n\n\t \* \n");

print\_matrix(B);

printf("\n\n\t = \n");

print\_matrix(C);

printf("\n\n");

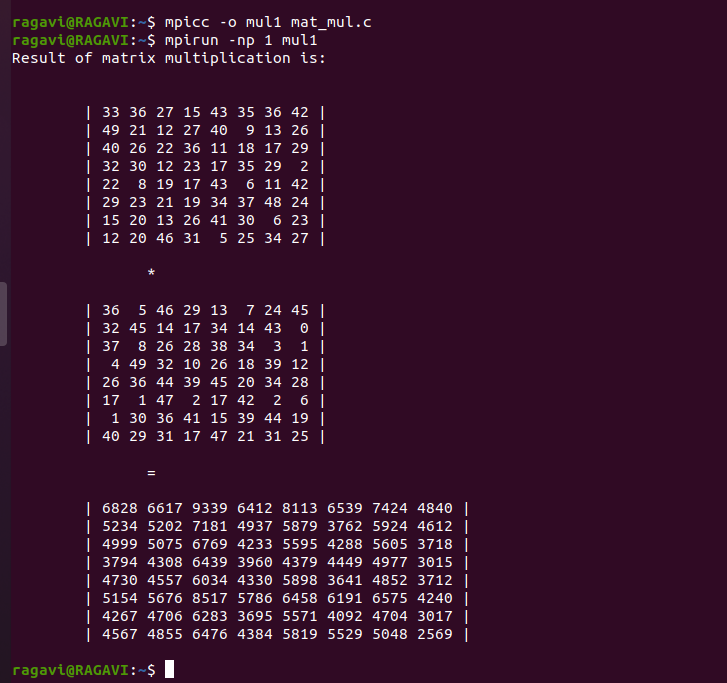
}

MPI\_Finalize();

return 0;

}

**Output:**

****