**Concurrent and Parallel Programming**

Assignment -5

Name- Ashutosh Soni

Id- 2018ucp1505

Q6: Implement parallel Quick sort using OpenMP, MPI, and Posix thread.

Ans:

**OpenMP**

**Program:**

// Quick Sort implementation using OpenMP

// Including header file

#include<bits/stdc++.h>

#include<omp.h>

using namespace std;

// Swapping two elements using pointer

void swap(int\* a, int\* b) {

int t = \*a;

\*a = \*b;

\*b = t;

}

// Partitioning of the array elements

int partition(int arr[],int start,int end){

// Declaration

int pivot = arr[end];

int i = (start - 1);

// Rearranging the array

for (int j = start; j <= end - 1; j++)

{

if (arr[j] < pivot)

{

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[end]);

// returning the respective index

return (i + 1);

}

// QuickSort Function using openmp

void quicksort(int arr[],int start,int end){

// Declaration

int index;

if(start<end){

// Getting the index of pivot by partitioning

index=partition(arr,start,end);

// Parallel sections

#pragma omp parallel sections

{

#pragma omp section

{

// Evaluating the left half

quicksort(arr,start,index-1);

}

#pragma omp section

{

// Evaluating the right half

quicksort(arr,index+1,end);

}

}

}

}

int main(){

// Declaration

int N;

// Taking input the number of elements we wants

cout<<"Enter the number of Elements you want to Enter"<<endl;

cin>>N;

// Declaration of array

int arr[N];

cout<<"Enter the array: "<<endl;

// taking input that array

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

cin>>arr[i];

}

// Calling quicksort having parallel code implementation

quicksort(arr,0,N-1);

// Printing the sorted array

cout<<"Array after Sorting is: "<<endl;

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

cout<<arr[i]<<" ";

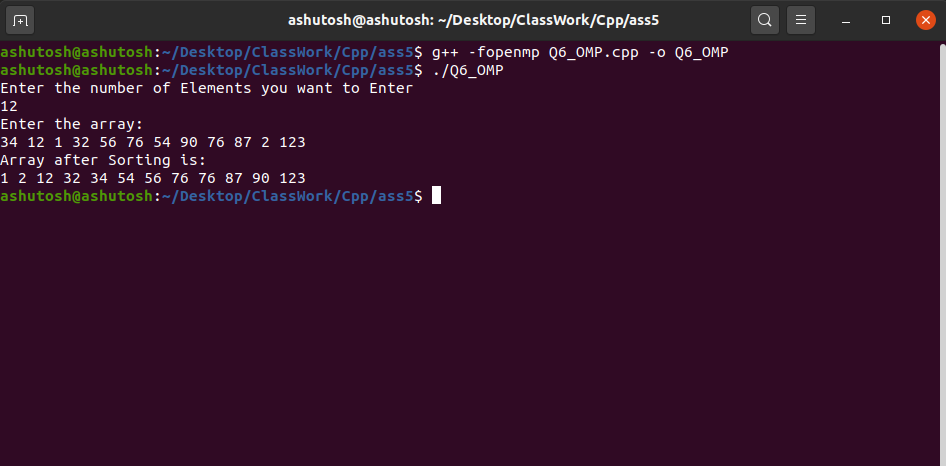
}

cout<<endl;

return 0;

}

**Output:**

****

**MPI**

**Program:**

// Quick Sort Using MPI

// Including header file

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

using namespace std;

// Swap two numbers with pointers

void swap(int \*arr, int i, int j)

{

int t = arr[i];

arr[i] = arr[j];

arr[j] = t;

}

// Quick sort for an array arr starting from index start and ending at index end

void quicksort(int \*arr, int start, int end)

{

// Declaration

int pivot, index;

// base case

if (end <= 1)

return;

// pick pivot and swap with first element

// Pivot is middle element

pivot = arr[start + end/2];

swap(arr, start, start + end/2);

// partitioning

index = start;

for (int i = start+1; i < start+end; i++)

if (arr[i] < pivot) {

index++;

swap(arr, i, index);

}

// swap pivot into place

swap(arr, start, index);

// recursive call for sorting of quick sort function

quicksort(arr, start, index-start);

quicksort(arr, index+1, start+end-index-1);

}

// Merge two arrays

int \* merge(int \*arr1, int n1, int \*arr2, int n2)

{

int \*result = (int \*)malloc((n1 + n2) \* sizeof(int));

int i = 0;

int j = 0;

int k;

for (k = 0; k < n1 + n2; k++) {

if (i >= n1) {

result[k] = arr2[j];

j++;

}

else if (j >= n2) {

result[k] = arr1[i];

i++;

}

else if (arr1[i] <arr2[j]) { // indices in bounds as i < n1 && j < n2

result[k] = arr1[i];

i++;

}

else { // v2[j] <= v1[i]

result[k] = arr2[j];

j++;

}

}

return result;

}

// Main Funciton starting....

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

{

int number\_of\_elements;

int \*data=NULL;

int chunk\_size,own\_chunk\_size;

int \*chunk;

FILE \*file=NULL;

double time\_taken;

MPI\_Status status;

if(argc!=3){

printf("Desired number of arguments are not thier in argv....\n");

printf("2 files required first one input and second one outpu....\n");

exit(-1);

}

int number\_of\_process,rank\_of\_process;

int rc=MPI\_Init(&argc,&argv);

if(rc!=MPI\_SUCCESS){

printf("Error in creating MPI program.\n Terminating......\n");

MPI\_Abort(MPI\_COMM\_WORLD,rc);

}

MPI\_Comm\_size(MPI\_COMM\_WORLD,&number\_of\_process);

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

if(rank\_of\_process==0){

// Opening the file

file=fopen(argv[1],"r");

// Printing Error message if any

if(file==NULL){

printf("Error in opening file\n");

exit(-1);

}

// Reading number of Elements in file ... First Value in file is number of Elements

printf("Reading number of Elements From file ....\n");

fscanf(file,"%d",&number\_of\_elements);

printf("Number of Elements in the file is %d \n",number\_of\_elements );

// Computing chunk size

chunk\_size= (number\_of\_elements%number\_of\_process==0) ? number\_of\_elements/number\_of\_process : number\_of\_elements/(number\_of\_process-1);

data = (int \*)malloc(number\_of\_process\*chunk\_size \* sizeof(int));

// Reading the rest elements in which operation is being performed

printf("Reading the array from the file.......\n");

for(int i=0;i<number\_of\_elements;i++){

fscanf(file,"%d",&data[i]);

}

// Padding data with zero

for(int i=number\_of\_elements;i<number\_of\_process\*chunk\_size;i++){

data[i]=0;

}

// Printing the array read from file

printf("Elements in the array is : \n");

for(int i=0;i<number\_of\_elements;i++){

printf("%d ",data[i]);

}

printf("\n");

fclose(file);

file=NULL;

}

// Blocks all process until reach this point

MPI\_Barrier(MPI\_COMM\_WORLD);

// Starts Timer

time\_taken -= MPI\_Wtime();

// BroadCast the Size to all the process from root process

MPI\_Bcast(&number\_of\_elements,1,MPI\_INT,0,MPI\_COMM\_WORLD);

// Computing chunk size

chunk\_size= (number\_of\_elements%number\_of\_process==0) ? number\_of\_elements/number\_of\_process : number\_of\_elements/(number\_of\_process-1);

// Calculating total size of chunk according to bits

chunk=(int \*)malloc(chunk\_size \* sizeof(int));

// Scatter the chuck size data to all process

MPI\_Scatter(data,chunk\_size,MPI\_INT,chunk,chunk\_size,MPI\_INT,0,MPI\_COMM\_WORLD);

free(data);

data=NULL;

// Compute size of own chunk and then sort them using quick sort

own\_chunk\_size = (number\_of\_elements >= chunk\_size\*(rank\_of\_process+1)) ? chunk\_size : (number\_of\_elements - chunk\_size\*rank\_of\_process);

// Sorting array with quick sort for every chunk as called by process

quicksort(chunk,0,own\_chunk\_size);

for(int step=1;step < number\_of\_process;step=2\*step){

if(rank\_of\_process%(2\*step) != 0){

MPI\_Send(chunk,own\_chunk\_size,MPI\_INT,rank\_of\_process-step,0,MPI\_COMM\_WORLD);

break;

}

if(rank\_of\_process+step < number\_of\_process){

int received\_chunk\_size = (number\_of\_elements >= chunk\_size\*(rank\_of\_process+ 2\*step)) ? (chunk\_size\*step) : (number\_of\_elements - chunk\_size\*(rank\_of\_process+step));

int \*chunk\_received;

chunk\_received= (int\*)malloc(received\_chunk\_size\*sizeof(int));

MPI\_Recv(chunk\_received,received\_chunk\_size,MPI\_INT,rank\_of\_process+step,0,MPI\_COMM\_WORLD,&status);

data = merge(chunk,own\_chunk\_size,chunk\_received,received\_chunk\_size);

free(chunk);

free(chunk\_received);

chunk=data;

own\_chunk\_size= own\_chunk\_size + received\_chunk\_size;

}

}

// Stop the timer

time\_taken+=MPI\_Wtime();

// Opening the other file as taken form input and writing it to the file and giving it as the output

if(rank\_of\_process==0){

// Opening the file

file= fopen(argv[2],"w");

if(file==NULL){

printf("Error in opening file... \n");

exit(-1);

}

// Printing total number of elements in the file

fprintf(file, "Total number of Elements in the array : %d\n",own\_chunk\_size);

// Printing the value of array in the file

for(int i=0;i<own\_chunk\_size;i++){

fprintf(file, "%d ",chunk[i]);

}

// Closing the file

fclose(file);

printf("\n\n\n\nResult printed in output.txt file and shown below: \n");

// For Printing in the terminal

printf("Total number of Elements given as input : %d\n", number\_of\_elements);

printf("Sorted array is: \n");

for(int i=0;i<number\_of\_elements;i++){

printf("%d ",chunk[i] );

}

printf("\n\nQuicksort %d ints on %d procs: %f secs\n", number\_of\_elements, number\_of\_process, time\_taken);

}

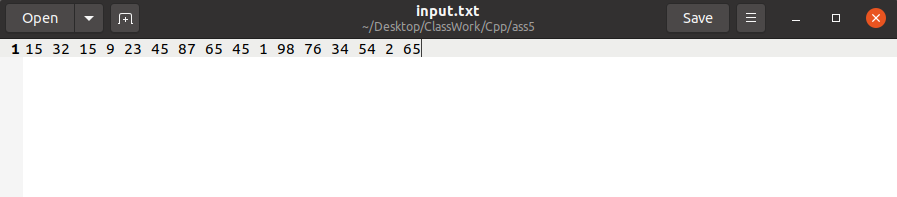
MPI\_Finalize();

return 0;

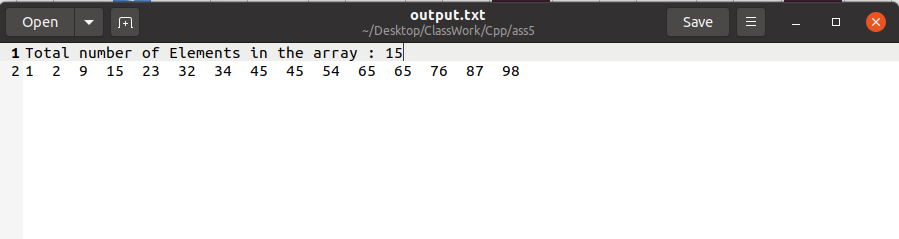
}

**Output:**

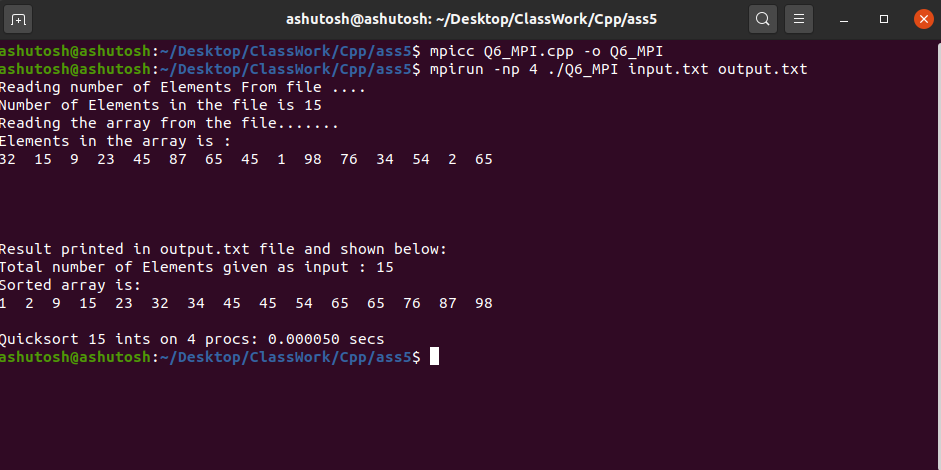
Input File:



Output File:



Terminal Output:



**Posix thread**

**Program:**

// Quick Sort Implementation using Posix Thread

// Including Header File

#include<bits/stdc++.h>

#include<pthread.h>

using namespace std;

// Structure

struct data\_set

{

int start\_index;

int end\_index;

int \*data;

};

// Swap function for swaping two Elements

void swap(int\* a, int\* b) {

int t = \*a;

\*a = \*b;

\*b = t;

}

// Partion funciton for making partition in array

int partition(int arr[],int left\_index,int right\_index){

// Declaration and initialization

// chosing pivot element form whcih we make partition

// Here pivot is last element of the array

int pivot= arr[right\_index];

int i=left\_index-1;

// Making array as per requirement

// arranging element smaller than pivot on left side and larger then pivot on right side

for(int j=left\_index;j<=right\_index-1;j++){

if(arr[j]<pivot){

i++;

swap(&arr[i],&arr[j]);

}

}

swap(&arr[i+1],&arr[right\_index]);

// Returing the partition index

return i+1;

}

// Quicksort Function for sorting array

void\* quick\_sort(void \*data){

// Retriving back the data sent from thread

struct data\_set \*info = (struct data\_set\*)data;

// Declaration of left index

int left\_index,right\_index,index;

// Initialization of left and right index

left\_index=info->start\_index;

right\_index=info->end\_index;

// Making recursive call of quick\_sort function

if(left\_index<right\_index){

// Declation of pthread and pthread attribute type object

pthread\_attr\_t attr;

pthread\_t first\_thread;

pthread\_t second\_thread;

// Making two pointers of type data\_set for making again call form thread

struct data\_set \*info1= new data\_set;

struct data\_set \*info2= new data\_set;

// Their initialization

info1->data=info->data;

info2->data=info->data;

// Initaization of pthread attribute

pthread\_attr\_init(&attr);

// For setting the set detach state of attribute

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

// Partitioning the array for recursive call

index = partition(info->data,left\_index,right\_index);

info1->start\_index=left\_index;

info1->end\_index=index-1;

// Creating pthread type object and printing the error if any

if(pthread\_create(&first\_thread,&attr,quick\_sort,info1)){

cout<<"Error in creating thread "<<endl;

// Exiting in case of not creation of thread

exit(-1);

}

info2->start\_index = index+1;

info2->end\_index = right\_index;

// Creating pthread type object and printing the error if any

if(pthread\_create(&second\_thread,&attr,quick\_sort,info2)){

cout<<"Error in craeting thread "<<endl;

// Exiting in case of not creation of thread

exit(-1);

}

// Joining the threads

pthread\_join(first\_thread,NULL);

pthread\_join(second\_thread,NULL);

}

return NULL;

}

int main(){

// Declaration of Number of threads

int N;

struct data\_set \*info = new data\_set;

// Taking number of elements as input

cout<<"Enter number of Elements in the Array : "<<endl;

cin>>N;

// Declaration of array

int A[N];

// Initialization of array

cout<<"Enter the array: "<<endl;

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

cin>>A[i];

}

// Initialization of structure of data\_set type

info->data=A;

info->start\_index=0;

info->end\_index=N-1;

// Declaration of pthread object

pthread\_t thread\_id;

// Creating and pthread object and printing the array of any

if(pthread\_create(&thread\_id,NULL,quick\_sort,info)){

cout<<"Error in creating thread"<<endl;

// Exit in case of error

exit(-1);

}

// Joining the pthread object

int r1=pthread\_join(thread\_id,NULL);

// Printing the array if any in case of joining

if(r1){

cout<<"Error in Joinging thread"<<endl;

// Exiting in case of error

exit(-1);

}

// Printing the array after sorting

cout<<"Sorted Array is: "<<endl;

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

cout<<A[i]<<" ";

}

cout<<endl;

// Exiting from pthread programming

pthread\_exit(NULL);

return 0;

}

**Output:**

