ASSIGNMENT NO 2

Title of the Assignment:

Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP

Problem Statement:

Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Useexisting algorithms and measure the performance of sequential and parallel algorithms.

Objective:

To offload parallel computations to the graphics card, when it is appropriate to do so, and to give some idea of how to think about code running in the massively parallel environment presented by today's graphics cards.

Outcome:

Students should understand the basic of OpenMP Programming Module using existingSorting techniques.

Prerequisites:

Data Structure and Files, Design and Analysis of Algorithm, OpenMP Tutorials.

Software Requirement:

Ubuntu 14.04, G Edit, GCC Complier.

Source Code:

```
#include<iostream>
#include<omp.h>
#include<bits/stdc++.h>
using namespace std;
void sequential_bubble_sort(int arr[],int size){
int array[size];
for(int i = 0 ; i < size; i++){
array[i] = arr[i];
}
double start = omp_get_wtime();
for(int i = 0; i < size - 1; i ++){
for(int j = 0; j < size - i - 1; j++){
if(array[j] > array[j+1]){
swap(array[j],array[j+1]);
}
}
}
double end = omp_get_wtime();
cout << "Sequential Bubble Sort:\n";
// for(int i = 0 ; i < size; i++){
// cout << array[i] << " ";
// }
cout << endl;
cout << "Time Required: " << end - start << endl;
}
void parallel_bubble_sort(int arr[],int size){
int array[size];
for(int i = 0 ; i < size; i++){
array[i] = arr[i];
}
```

```
double start = omp_get_wtime();
for(int k = 0; k < size;k ++){
if(k % 2 == 0){
#pragma omp parallel for
for(int i = 1; i < size - 1; i += 2){
if(array[i] > array[i+1]){
swap(array[i],array[i+1]);
}
}
}
else{
#pragma omp parallel for
for(int i = 0; i < size - 1; i += 2){
if(array[i] > array[i+1]){
swap(array[i],array[i+1]);
}
}
}
}
double end = omp_get_wtime();
cout << "Parallel Bubble Sort:\n";
// for(int i = 0 ; i < size; i++){
// cout << array[i] << " ";
// }
cout << endl;
cout << "Time Required: " << end - start << endl;
}
void merge(int array[],int low, int mid, int high,int size){
int temp[size];
int i = low;
int j = mid + 1;
```

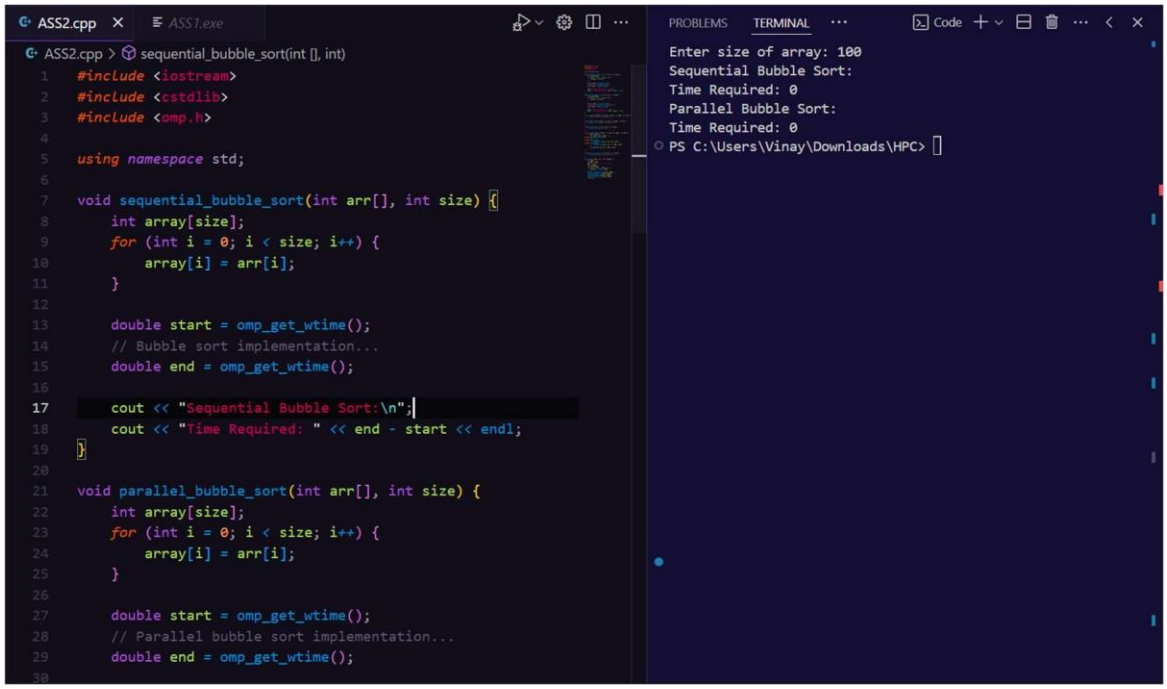
```
int k = 0;
while((i <= mid) && (j <= high)){
if(array[i] >= array[j]){
temp[k] = array[j];
k++;
j++;
}
else{
temp[k] = array[i];
k++;
i++;
}
}
while(i <= mid){
temp[k] = array[i];
k++;
i++;
}
while(j <= high){
temp[k] = array[j];
k++;
j++;
}
k = 0;
for(int i = low;i <= high;i++){
array[i] = temp[k];
k++;
}
}
void mergesort(int array[],int low,int high,int size){
if(low < high){
```

```
int mid = (low + high) / 2;
mergesort(array,low,mid,size);
mergesort(array,mid+1,high,size);
merge(array,low,mid,high,size);
}
}
void perform_merge_sort(int arr[],int size){
int array[size];
for(int i = 0 ; i < size; i++){
array[i] = arr[i];
}
double start = omp_get_wtime();
mergesort(array,0,size-1,size);
double end = omp_get_wtime();
cout << "Merge Sort:\n";
// for(int i = 0 ; i < size; i++){
// cout << array[i] << " ";
// }
cout << endl;
cout << "Time Required: " << end - start << endl;
}
void p_mergesort(int array[],int low,int high,int size){
if(low < high){
int mid = (low + high) / 2;
#pragma omp parallel sections
{
#pragma omp section
p_mergesort(array,low,mid,size);
#pragma omp section
p_mergesort(array,mid+1,high,size);
}
```

```
merge(array,low,mid,high,size);
}
}
void perform_p_merge_sort(int arr[],int size){
int array[size];
for(int i = 0 ; i < size; i++){
array[i] = arr[i];
}
double start = omp_get_wtime();
p_mergesort(array,0,size-1,size);
double end = omp_get_wtime();
cout << "Parallel Merge Sort:\n";
// for(int i = 0 ; i < size; i++){
// cout << array[i] << " ";
// }
cout << endl;
cout << "Time Required: " << end - start << endl;
}
int main(int argc, char const *argv[])
{
int SIZE;
int MAX = 1000;
cout << "Enter size of array: ";
cin >> SIZE;
int array[SIZE];
for(int i = 0 ; i < SIZE; i ++){
array[i] = rand() % MAX;
}
// cout << "Initial Array:\n";
// for(int i = 0 ; i < SIZE; i++){
// cout << array[i] << " ";
```

```
// }
cout << endl;
sequential_bubble_sort(array,SIZE);
parallel_bubble_sort(array,SIZE);
perform_merge_sort(array,SIZE);
perform_p_merge_sort(array,SIZE);
return 0;
}
```

OUTPUT:



Conclusion:

We have tested both the Sorting algorithms for different number elements with respective to time. As the number of elements increased time required for OpenMP is reduced.