Assignment – 2

Code:

```
#include <bits/stdc++.h>
#include <ctime>
#include <omp.h>
using namespace std;

// Function to perform sequential bubble sort
void bubbleSortSeq(vector<int>& arr)
{
   int n = arr.size();
   for (int i = 0; i < n-1; i++)
   {
      for (int j = 0; j < n-i-1; j++)
      {
         if (arr[j] > arr[j+1]) swap(arr[j], arr[j+1]);
      }
   }

   cout << "First 10 values in array after Sequential Bubble sort: ";
   for(int i =0; i<10; i++)
   {
      cout << arr[i] << " ";
   }
   cout << endl;
}

// Function to perform parallel bubble sort
void bubbleSortPar(vector<int>& arr)
{
   int n = arr.size();
   #pragma omp parallel
   {
      for (int i = 0; i < n-1; i++)
      {
         #pragma omp for
         for (int j = 0; j < n-i-1; j++)
         {
            if (arr[j] > arr[j+1]) swap(arr[j], arr[j+1]);
         }
```

```
        }
    }

    cout << "\nFirst 10 values in array after Parallel Bubble sort: ";
    for(int i =0; i<10; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// Function to merge two sorted subarrays
void merge(vector<int>& arr, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
```

```
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Function to perform sequential merge sort
void mergeSortSeq(vector<int>& arr, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        mergeSortSeq(arr, left, middle);
        mergeSortSeq(arr, middle + 1, right);

        merge(arr, left, middle, right);
    }

}

// Function to merge two sorted subarrays in parallel
void parallelMerge(vector<int>& arr, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    vector<int> L(n1), R(n2);

    #pragma omp parallel for
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    #pragma omp parallel for
    for (int j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];

    int i = 0, j = 0, k = left;
    #pragma omp parallel sections
    {
        #pragma omp section
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
```

```
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    #pragma omp section
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    #pragma omp section
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
  }
}



// Function to perform parallel merge sort
void mergeSortPar(vector<int>& arr, int left, int right) {
  if (left < right) {
    int middle = left + (right - left) / 2;

    #pragma omp parallel sections
    {
      #pragma omp section
      mergeSortPar(arr, left, middle);
      #pragma omp section
      mergeSortPar(arr, middle + 1, right);
    }

    merge(arr, left, middle, right);
```

```
    }

}

// Function to merge two sorted subarrays in parallel
void parallelMerge(vector<int>& arr, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    vector<int> L(n1), R(n2);

    #pragma omp parallel for
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    #pragma omp parallel for
    for (int j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];

    int i = 0, j = 0, k = left;
    #pragma omp parallel sections
    {
        #pragma omp section
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        #pragma omp section
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        #pragma omp section
```

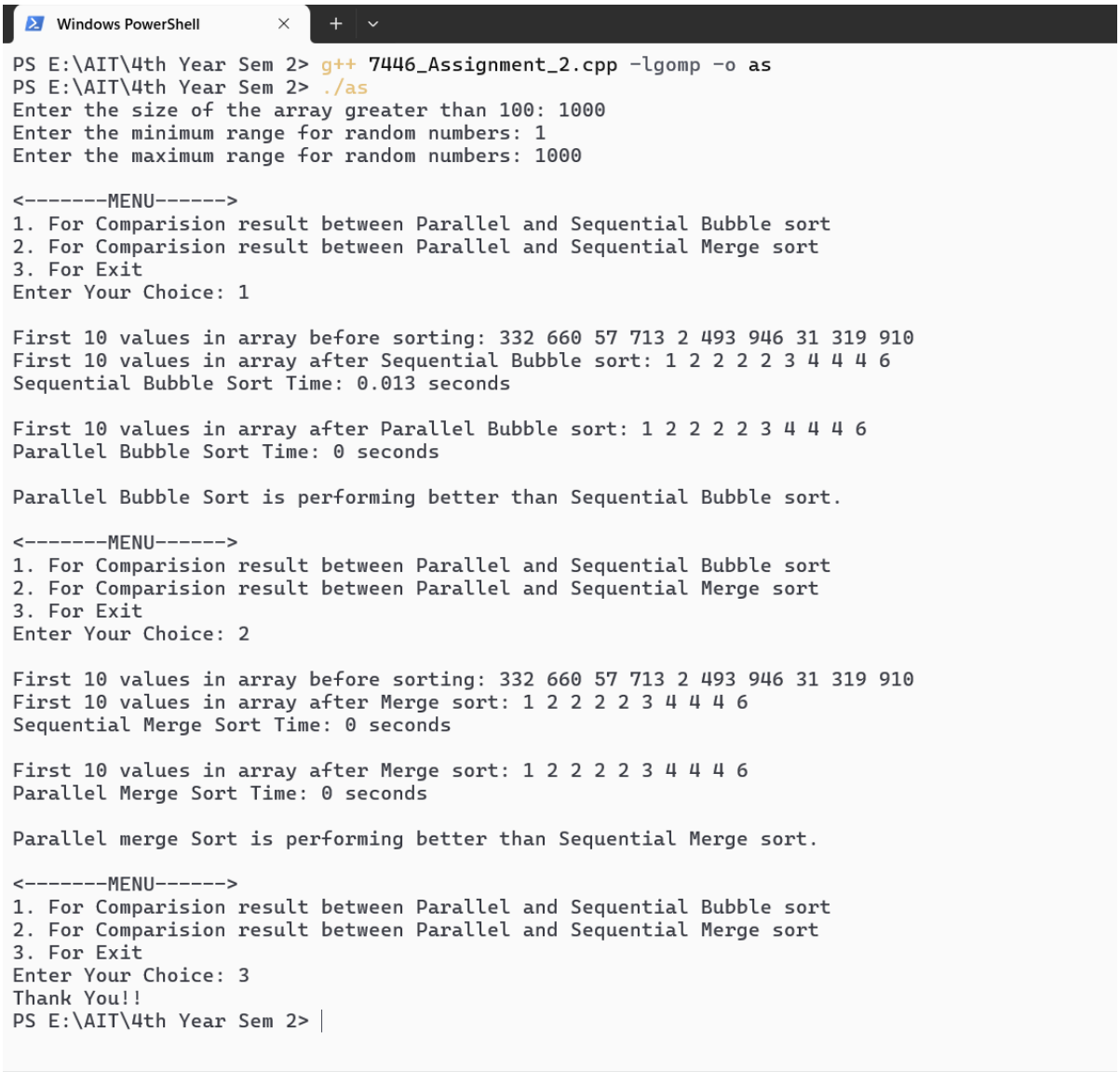
```
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
}

// Function to measure execution time of sorting algorithms
double measureTime(void (*sortFunction)(vector<int>&), vector<int>& arr) {
    double start = omp_get_wtime();
    sortFunction(arr);
    double end = omp_get_wtime();
    return end - start;
}

// Function to measure execution time of sorting algorithms with parameters
double measuretime(void (*sortFunction)(vector<int>&, int, int), vector<int>& arr, int left, int
right) {
    double start = omp_get_wtime();
    sortFunction(arr, left, right);
    double end = omp_get_wtime();
    return end - start;
}

// Main function
signed main() {
    int size;
    int min_range, max_range;

    cout << "Enter the size of the array greater than 100: ";
    cin >> size;
    cout << "Enter the minimum range for random numbers: ";
    cin >> min_range;
    cout << "Enter the maximum range for random numbers: ";
    cin >> max_range;

    vector<int> arr(size);

    srand(time(0));
    for (int i = 0; i < size; ++i) {
```

```
    arr[i] = rand() % (max_range - min_range + 1) + min_range;
  }

  bool flag = true;
  while(flag)
  {
    cout << "\n<-------MENU------>"<<endl;
    cout << "1. For Comparision result between Parallel and Sequential Bubble sort" << endl;
    cout << "2. For Comparision result between Parallel and Sequential Merge sort" << endl;
    cout << "3. For Exit"<<endl;
    int ch;
    cout << "Enter Your Choice: ";
    cin >> ch;

    switch(ch)
    {
      case 1:
      {
        vector<int> arr_bubble = arr;
        cout << "\nFirst 10 values in array before sorting: ";
        for(int i =0; i<10; i++)
        {
          cout << arr[i] << " ";
        }
        cout << endl;

        double sequential_bubble_time = measureTime(bubbleSortSeq, arr_bubble);
        cout << "Sequential Bubble Sort Time: " << sequential_bubble_time << " seconds" <<
endl;


        vector<int> arr_parallel_bubble = arr;

        double parallel_bubble_time = measureTime(bubbleSortPar, arr_parallel_bubble);
        cout << "Parallel Bubble Sort Time: " << parallel_bubble_time << " seconds" << endl;

        if(sequential_bubble_time < parallel_bubble_time) cout << "\nSequential Bubble Sort
is performing better than Parallel Bubble sort." << endl;
        else cout << "\nParallel Bubble Sort is performing better than Sequential Bubble sort."
<< endl;
```

```
            break;
        }
        case 2:
        {
            vector<int> arr_merge = arr;
            cout << "\nFirst 10 values in array before sorting: ";
            for(int i =0; i<10; i++)
            {
                cout << arr[i] << " ";
            }
            cout << endl;

            double sequential_merge_time = measuretime(mergeSortSeq, arr_merge, 0, size - 1);

            cout << "First 10 values in array after Merge sort: ";
            for(int i =0; i<10; i++)
            {
                cout << arr_merge[i] << " ";
            }
            cout << endl;
            cout << "Sequential Merge Sort Time: " << sequential_merge_time << " seconds" <<
endl;

            vector<int> arr_parallel_merge = arr;
            double parallel_merge_time = measuretime(mergeSortPar, arr_parallel_merge,0, size -
            1);

            cout << "\nFirst 10 values in array after Merge sort: ";
            for(int i =0; i<10; i++)
            {
                cout << arr_parallel_merge[i] << " ";
            }
            cout << endl;
            cout << "Parallel Merge Sort Time: " << parallel_merge_time << " seconds" << endl;

            if(sequential_merge_time < parallel_merge_time)
                    cout << "\nSequential Merge Sort is performing better than Parallel Merge sort."
                    << endl;
            else
                    cout << "\nParallel merge Sort is performing better than Sequential Merge sort."
                    << endl;
```

```
            break;}
        case 3:
        {
            cout << "Thank You!!" << endl;
            flag = false;
            break;
        }

        default:
        {
            cout << "Invalid choice, Try again" << endl;
            break;
        }
      }
   }
   return 0;
}
```

Output:


Windows PowerShell
PS E:\AIT\4th Year Sem 2> g++ 7446_Assignment_2.cpp -lgomp -o as
PS E:\AIT\4th Year Sem 2> ./as
Enter the size of the array greater than 100: 1000
Enter the minimum range for random numbers: 1
Enter the maximum range for random numbers: 1000
<-------MENU------>
1. For Comparision result between Parallel and Sequential Bubble sort
2. For Comparision result between Parallel and Sequential Merge sort
3. For Exit
Enter Your Choice: 1
First 10 values in array before sorting: 332 660 57 713 2 493 946 31 319 910
First 10 values in array after Sequential Bubble sort: 1 2 2 2 2 3 4 4 4 6
Sequential Bubble Sort Time: 0.013 seconds
First 10 values in array after Parallel Bubble sort: 1 2 2 2 2 3 4 4 4 6
Parallel Bubble Sort Time: 0 seconds
Parallel Bubble Sort is performing better than Sequential Bubble sort.
<-------MENU------>
1. For Comparision result between Parallel and Sequential Bubble sort
2. For Comparision result between Parallel and Sequential Merge sort
3. For Exit
Enter Your Choice: 2
First 10 values in array before sorting: 332 660 57 713 2 493 946 31 319 910
First 10 values in array after Merge sort: 1 2 2 2 2 3 4 4 4 6
Sequential Merge Sort Time: 0 seconds
First 10 values in array after Merge sort: 1 2 2 2 2 3 4 4 4 6
Parallel Merge Sort Time: 0 seconds
Parallel merge Sort is performing better than Sequential Merge sort.
<-------MENU------>
1. For Comparision result between Parallel and Sequential Bubble sort
2. For Comparision result between Parallel and Sequential Merge sort
3. For Exit
Enter Your Choice: 3
Thank You!!
PS E:\AIT\4th Year Sem 2>