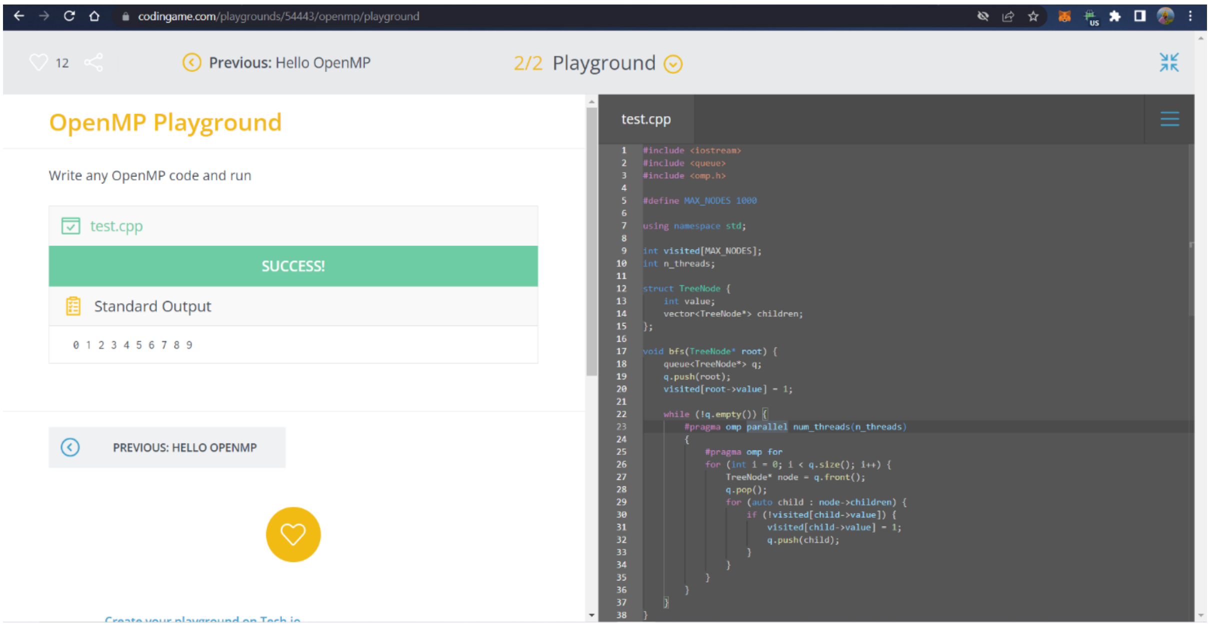
Experiment No:01

Title: Design and implement Parallel Breadth First Search based on existing algorithms using open MP use a tree or an undirected graph for BFS.

```
#include<iostream>
#include <queue>
#include <omp.h>

#define MAX_NODES 1000

using namespace std;

int visited[MAX_NODES];
int n_threads;

struct TreeNode {
    int value;
    vector<TreeNode*> children;
};

void bfs(TreeNode* root) {
    queue<TreeNode*> q;
    q.push(root);
    visited[root->value] = 1;

    while (!q.empty()) {
        #pragma omp parallel num_threads(n_threads)
        {
            #pragma omp for
            for (int i = 0; i < q.size(); i++) {
                TreeNode* node = q.front();
                q.pop();
                for (auto child : node->children) {
                    if (!visited[child->value]) {
                        visited[child->value] = 1;
                        q.push(child);
                    }
                }
            }
        }
    }
}

int main() {
```

```
    // Construct a tree
    TreeNode* root = new TreeNode{0};
    for (int i = 1; i < 10; i++) {
        TreeNode* child = new TreeNode{i};
        root->children.push_back(child);
        for (int j = 0; j < i; j++) {
            TreeNode* grandchild = new TreeNode{j};
            child->children.push_back(grandchild);
        }
    }

    // Initialize OpenMP
    n_threads = omp_get_max_threads();

    // Run BFS
    bfs(root);

    // Print visited nodes
    for (int i = 0; i < 10; i++) {
        if (visited[i]) {
            cout << i << " ";
        }
    }
    cout << endl;

    return 0;
}
```

OUTPUT→

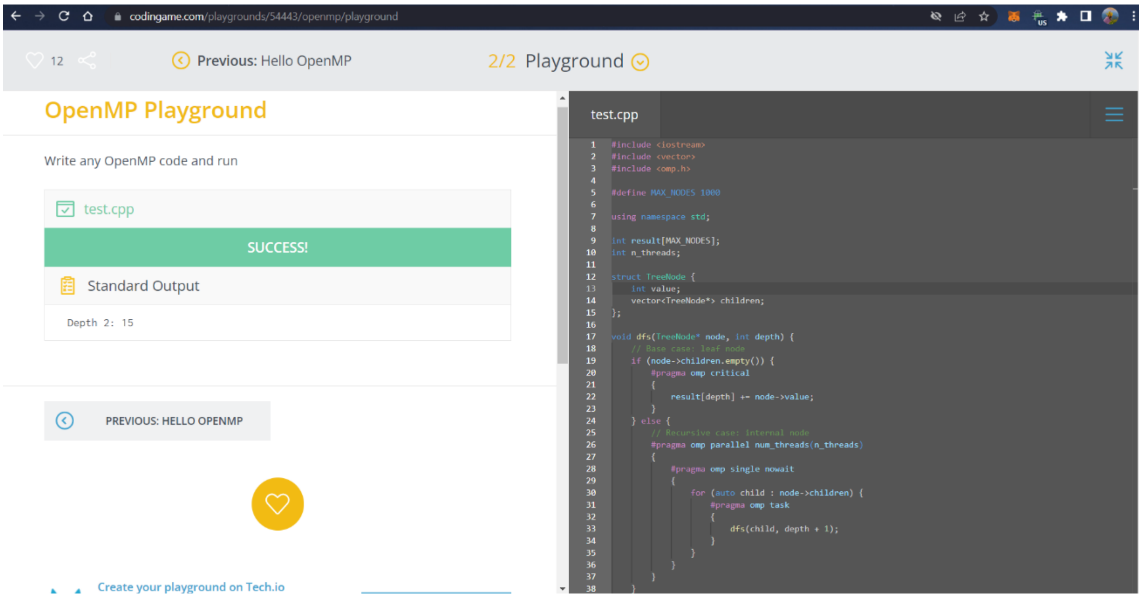


Title: Design and implement Parallel Depth First Search based on existing algorithms using open MP use a tree or an undirected graph for DFS.

```
#include <iostream>
#include <vector>
#include <omp.h>

#define MAX_NODES 1000

using namespace std;

int result[MAX_NODES];
int n_threads;

struct TreeNode {
   int value;
   vector<TreeNode*> children;
};

void dfs(TreeNode* node, int depth) {
   // Base case: leaf node
   if (node->children.empty()) {
      #pragma omp critical
      {
         result[depth] += node->value;
      }
   } else {
      // Recursive case: internal node
      #pragma omp parallel num_threads(n_threads)
      {
```

```
        #pragma omp single nowait
        {
          for (auto child : node->children) {
            #pragma omp task
            {
              dfs(child, depth + 1);
            }
          }
        }
      }
    }
}

int main() {
    // Construct a tree
    TreeNode* root = new TreeNode{1};
    TreeNode* child1 = new TreeNode{2};
    TreeNode* child2 = new TreeNode{3};
    root->children.push_back(child1);
    root->children.push_back(child2);
    TreeNode* grandchild1 = new TreeNode{4};
    TreeNode* grandchild2 = new TreeNode{5};
    TreeNode* grandchild3 = new TreeNode{6};
    child1->children.push_back(grandchild1);
    child2->children.push_back(grandchild2);
    child2->children.push_back(grandchild3);

    // Initialize OpenMP
    n_threads = omp_get_max_threads();

    // Run DFS
    dfs(root, 0);

    // Print results
    for (int i = 0; i < MAX_NODES; i++) {
        if (result[i] > 0) {
            cout << "Depth " << i << ": " << result[i] << endl;
        }
    }

    return 0;
}
```

OUTPUT→


codingame.com/playgrounds/54443/openmp/playground
12
Previous: Hello OpenMP
2/2 Playground
OpenMP Playground
Write any OpenMP code and run
test.cpp
SUCCESS!
Standard Output
Depth 2: 15
PREVIOUS: HELLO OPENMP
Create your playground on Tech.io
test.cpp
#include <iostream>
#include <vector>
#include <omp.h>
#define MAX_NODES 1000
using namespace std;
int result[MAX_NODES];
int n_threads;
struct TreeNode {
int value;
vector<TreeNode*> children;
};
void dfs(TreeNode* node, int depth) {
// Base case: leaf node
if (node->children.empty()) {
#pragma omp critical
{
result[depth] += node->value;
}
} else {
// Recursive case: internal node
#pragma omp parallel num_threads(n_threads)
{
#pragma omp single nowait
{
for (auto child : node->children) {
#pragma omp task
{
dfs(child, depth + 1);
}
}
}
}
}

Experiment No:02

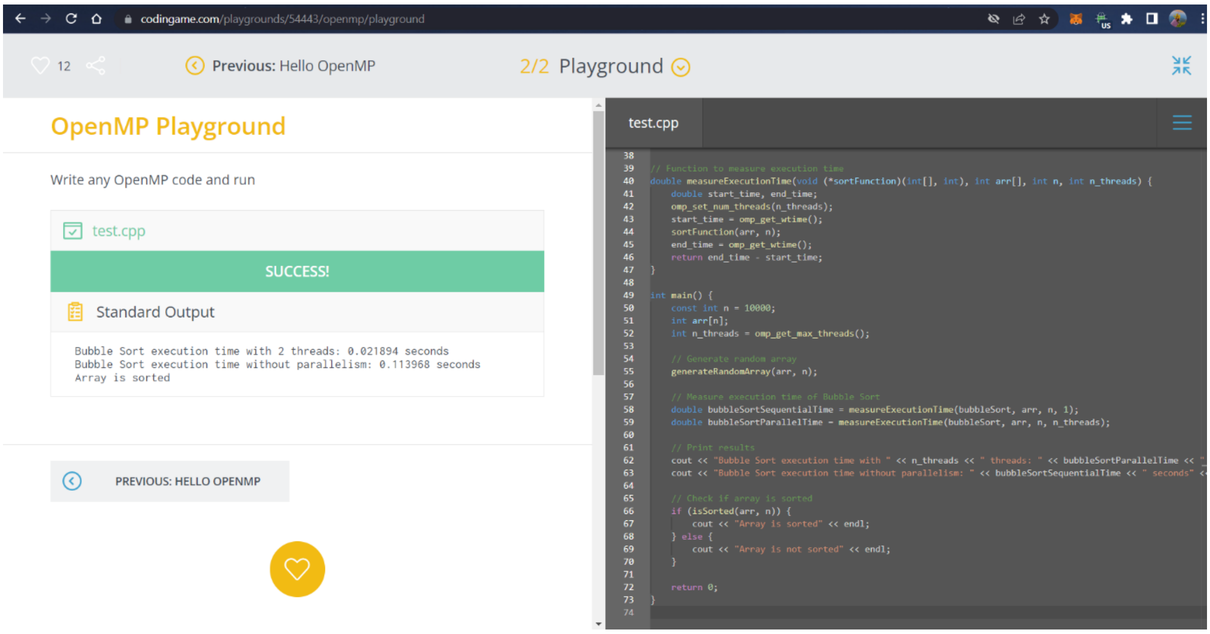
Title: Write a program to implement Bubble sort and merge sort using open MP ,use existing algorithm and measure performance of sequential and parallel algorithms.

Bubble Sort:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>

using namespace std;

// Bubble Sort algorithm
void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n-1; i++) {
      for (int j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
}

// Function to generate random array
void generateRandomArray(int arr[], int n) {
   srand(time(NULL));
   for (int i = 0; i < n; i++) {
      arr[i] = rand() % 100;
   }
}

// Function to check if array is sorted
bool isSorted(int arr[], int n) {
   for (int i = 0; i < n-1; i++) {
      if (arr[i] > arr[i+1]) {
         return false;
      }
   }
   return true;
}

// Function to measure execution time
```

```
double measureExecutionTime(void (*sortFunction)(int[], int), int arr[], int n, int n_threads)
{
   double start_time, end_time;
   omp_set_num_threads(n_threads);
   start_time = omp_get_wtime();
   sortFunction(arr, n);
   end_time = omp_get_wtime();
   return end_time – start_time;
}

int main() {
   const int n = 10000;
   int arr[n];
   int n_threads = omp_get_max_threads();

   // Generate random array
   generateRandomArray(arr, n);

   // Measure execution time of Bubble Sort
   double bubbleSortSequentialTime = measureExecutionTime(bubbleSort, arr, n, 1);
   double bubbleSortParallelTime = measureExecutionTime(bubbleSort, arr, n, n_threads);

   // Print results
   cout << "Bubble Sort execution time with " << n_threads << " threads: " <<
bubbleSortParallelTime << " seconds" << endl;
   cout << "Bubble Sort execution time without parallelism: " << bubbleSortSequentialTime
<< " seconds" << endl;

   // Check if array is sorted
   if (isSorted(arr, n)) {
      cout << "Array is sorted" << endl;
   } else {
      cout << "Array is not sorted" << endl;
   }

   return 0;
}
```

OUTPUT→



Merge Sort:

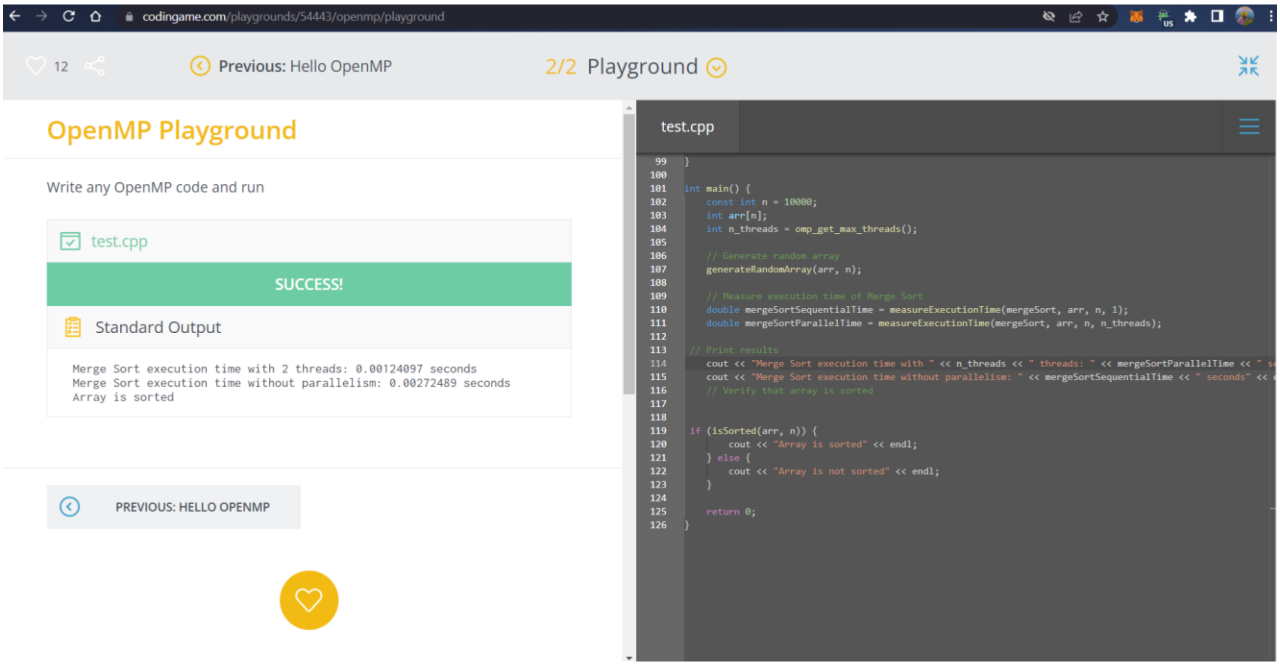
```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>

using namespace std;

// Merge Sort algorithm
void merge(int arr[], int left, int mid, int right) {
   int i, j, k;
   int n1 = mid – left + 1;
   int n2 = right – mid;

   // create temporary arrays
   int L[n1], R[n2];

   // copy data to temporary arrays L[] and R[]
   for (i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];

   // merge the temporary arrays back into arr[left..right]
   i = 0; // initial index of first subarray
   j = 0; // initial index of second subarray
```

```
  k = left; // initial index of merged subarray
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    }
    else {
      arr[k] = R[j];
      j++;
    }
    k++;
  }

  // copy the remaining elements of L[], if any
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  // copy the remaining elements of R[], if any
  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
  }
}

void mergeSort(int arr[], int left, int right) {
  if (left < right) {
    int mid = left + (right – left) / 2;

    // parallelize the recursive calls to mergeSort
    #pragma omp parallel sections
    {
      #pragma omp section
      mergeSort(arr, left, mid);

      #pragma omp section
      mergeSort(arr, mid + 1, right);
    }

    // merge the two sorted halves
    merge(arr, left, mid, right);
  }
}
```

```cpp
// Function to generate random array
void generateRandomArray(int arr[], int n) {
   srand(time(NULL));
   for (int i = 0; i < n; i++) {
      arr[i] = rand() % 100;
   }
}

// Function to check if array is sorted
bool isSorted(int arr[], int n) {
   for (int i = 0; i < n-1; i++) {
      if (arr[i] > arr[i+1]) {
         return false;
      }
   }
   return true;
}

// Function to measure execution time
double measureExecutionTime(void (*sortFunction)(int[], int, int), int arr[], int n, int
n_threads) {
   double start_time, end_time;
   omp_set_num_threads(n_threads);
   start_time = omp_get_wtime();
   sortFunction(arr, 0, n-1);
   end_time = omp_get_wtime();
   return end_time – start_time;
}

int main() {
   const int n = 10000;
   int arr[n];
   int n_threads = omp_get_max_threads();

   // Generate random array
   generateRandomArray(arr, n);

   // Measure execution time of Merge Sort
   double mergeSortSequentialTime = measureExecutionTime(mergeSort, arr, n, 1);
   double mergeSortParallelTime = measureExecutionTime(mergeSort, arr, n, n_threads);

 // Print results
   cout << “Merge Sort execution time with “ << n_threads << “ threads: “ <<
mergeSortParallelTime << “ seconds” << endl;
   cout << “Merge Sort execution time without parallelism: “ << mergeSortSequentialTime
<< “ seconds” << endl;
   // Verify that array is sorted
```

```cpp
if (isSorted(arr, n)) {
        cout << "Array is sorted" << endl;
    } else {
        cout << "Array is not sorted" << endl;
    }

    return 0;
}
```

OUTPUT→



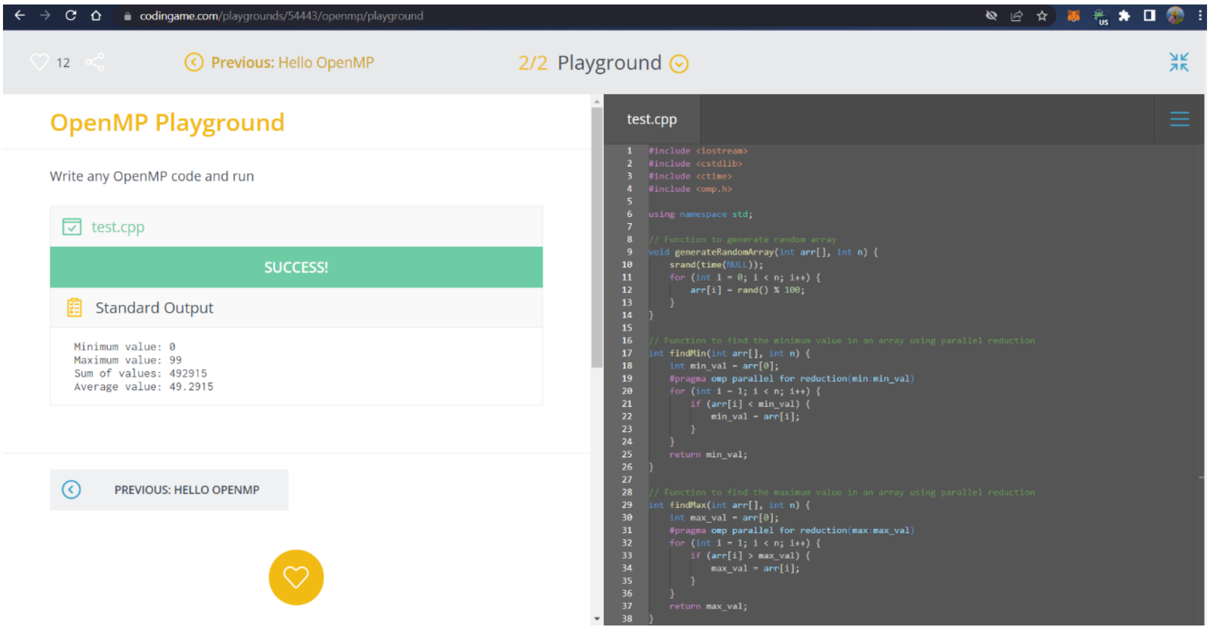
Experiment No:03

Title:Implement Min, Max ,Sum and Average operations using Parallel Reduction.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>

using namespace std;

// Function to generate random array
void generateRandomArray(int arr[], int n) {
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
}

// Function to find the minimum value in an array using parallel reduction
int findMin(int arr[], int n) {
    int min_val = arr[0];
    #pragma omp parallel for reduction(min:min_val)
    for (int i = 1; i < n; i++) {
        if (arr[i] < min_val) {
            min_val = arr[i];
        }
    }
    return min_val;
}

// Function to find the maximum value in an array using parallel reduction
int findMax(int arr[], int n) {
    int max_val = arr[0];
    #pragma omp parallel for reduction(max:max_val)
    for (int i = 1; i < n; i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    return max_val;
}

// Function to find the sum of values in an array using parallel reduction
int findSum(int arr[], int n) {
```

```
    int sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}

// Function to find the average value in an array using parallel reduction
double findAverage(int arr[], int n) {
    double avg = 0;
    #pragma omp parallel for reduction(+:avg)
    for (int i = 0; i < n; i++) {
        avg += arr[i];
    }
    avg /= n;
    return avg;
}

int main() {
    const int n = 10000;
    int arr[n];

    // Generate random array
    generateRandomArray(arr, n);

    // Find minimum value
    int min_val = findMin(arr, n);
    cout << "Minimum value: " << min_val << endl;

    // Find maximum value
    int max_val = findMax(arr, n);
    cout << "Maximum value: " << max_val << endl;

    // Find sum of values
    int sum = findSum(arr, n);
    cout << "Sum of values: " << sum << endl;

    // Find average value
    double avg = findAverage(arr, n);
    cout << "Average value: " << avg << endl;

    return 0;
}
```

OUTPUT→

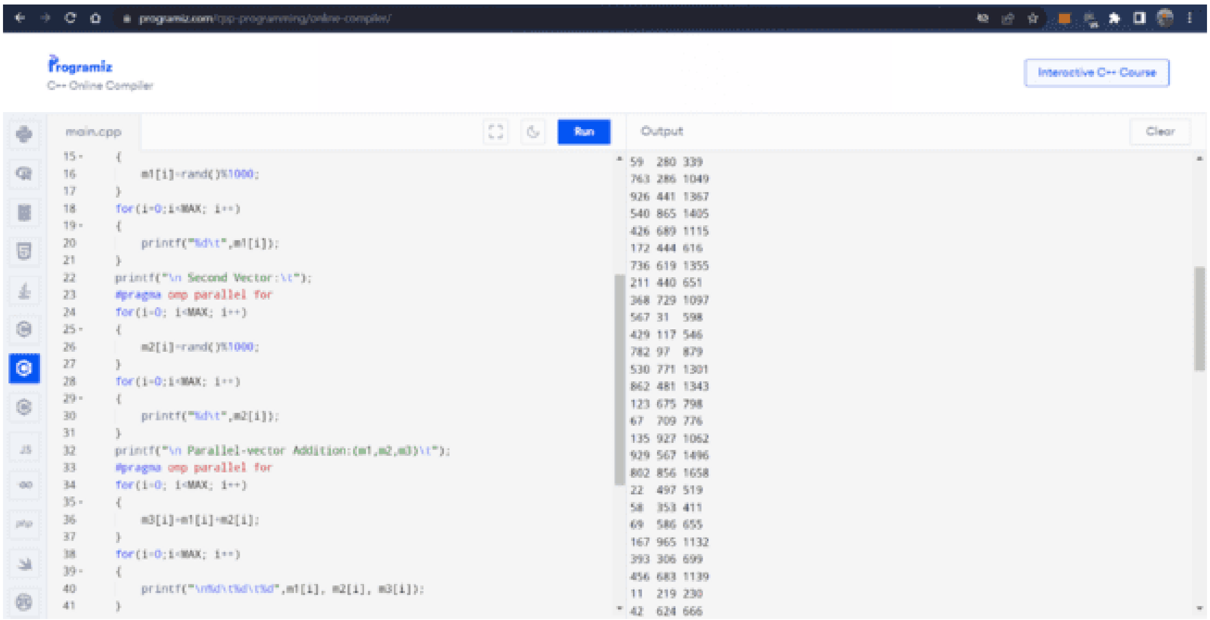

codingame.com/playgrounds/54443/openmp/playground
12
Previous: Hello OpenMP
2/2 Playground
OpenMP Playground
Write any OpenMP code and run
test.cpp
SUCCESS!
Standard Output
Minimum value: 0
Maximum value: 99
Sum of values: 492915
Average value: 49.2915
PREVIOUS: HELLO OPENMP
test.cpp

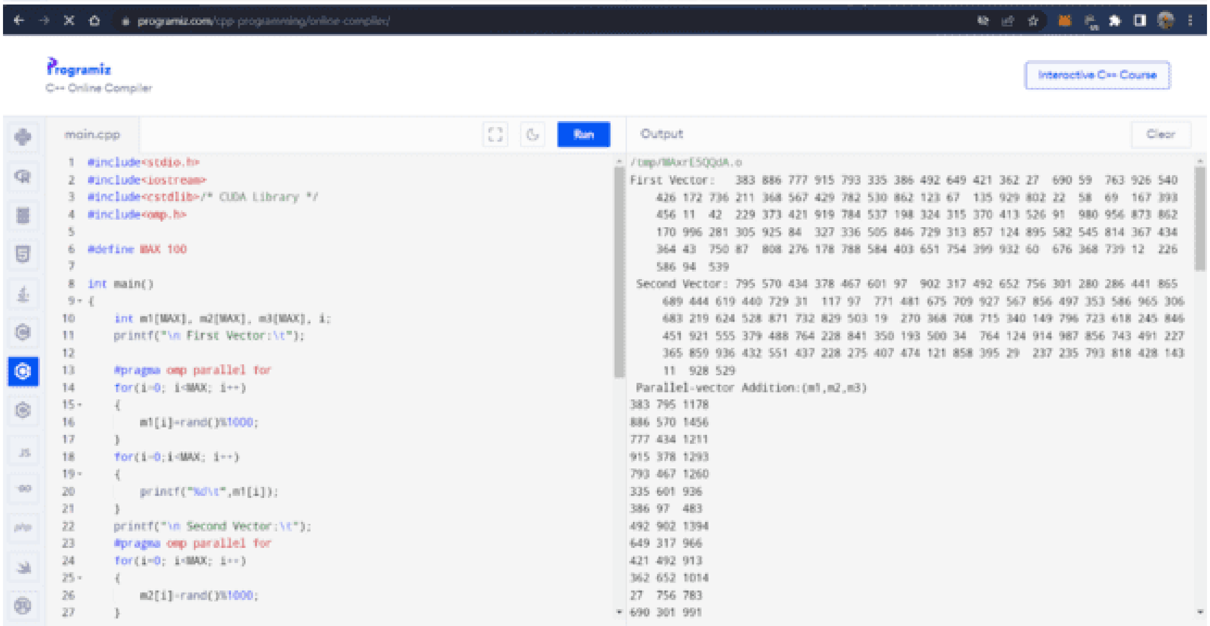
Experiment No:4

Addition of large vectors:-

```
#include<stdio.h>
#include<iostream>
#include<cstdlib>/* CUDA Library */
#include<omp.h>

#define MAX 100

int main()
{
    int m1[MAX], m2[MAX], m3[MAX], i;
    printf("\n First Vector:\t");

    #pragma omp parallel for
    for(i=0; i<MAX; i++)
    {
        m1[i]=rand()%1000;
    }
    for(i=0;i<MAX; i++)
    {
        printf("%d\t",m1[i]);
    }
    printf("\n Second Vector:\t");
    #pragma omp parallel for
    for(i=0; i<MAX; i++)
    {
        m2[i]=rand()%1000;
    }
    for(i=0;i<MAX; i++)
    {
        printf("%d\t",m2[i]);
    }
    printf("\n Parallel-vector Addition:(m1,m2,m3)\t");
    #pragma omp parallel for
    for(i=0; i<MAX; i++)
    {
        m3[i]=m1[i]+m2[i];
    }
    for(i=0;i<MAX; i++)
    {
        printf("\n%d\t%d\t%d",m1[i], m2[i], m3[i]);
    }
}
```

OUTPUT→





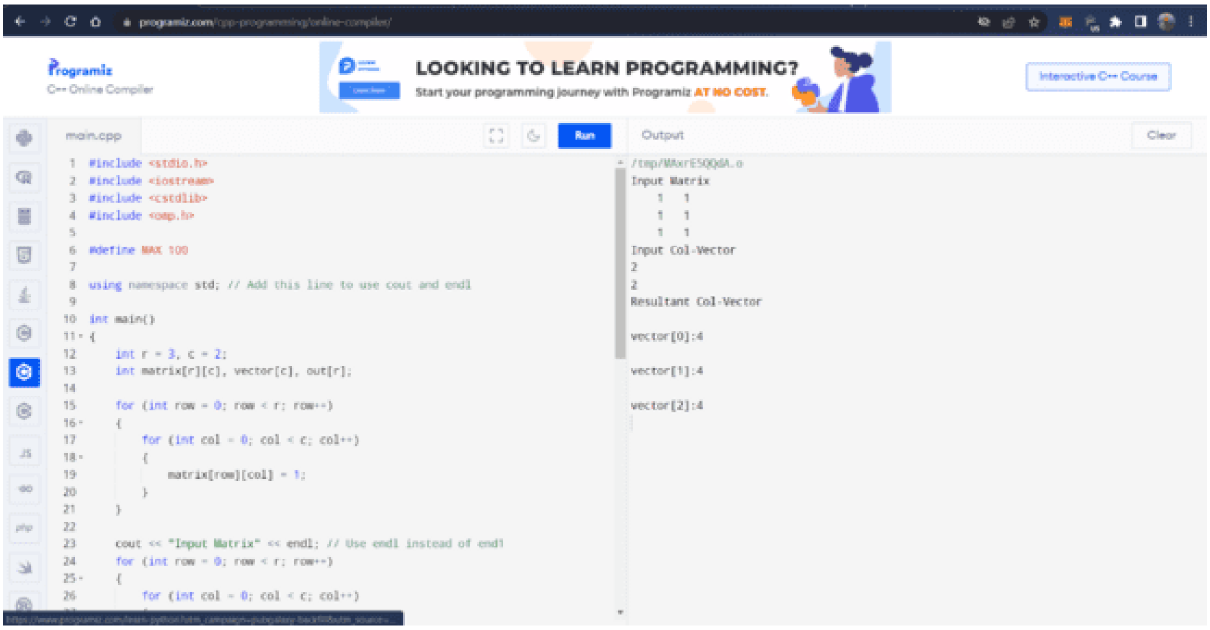
Matrix multiplication using CUDA:-

```
#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include <omp.h>

#define MAX 100

using namespace std; // Add this line to use cout and endl
```

```
int main()
{
    int r = 3, c = 2;
    int matrix[r][c], vector[c], out[r];

    for (int row = 0; row < r; row++)
    {
        for (int col = 0; col < c; col++)
        {
            matrix[row][col] = 1;
        }
    }

    cout << "Input Matrix" << endl; // Use endl instead of end1
    for (int row = 0; row < r; row++)
    {
        for (int col = 0; col < c; col++)
        {
            cout << "\t" << matrix[row][col];
        }
        cout << "" << endl; // Use endl instead of end1
    }

    for (int col = 0; col < c; col++) // Change row to col
    {
        vector[col] = 2;
    }
    cout << "Input Col-Vector" << endl; // Use endl instead of end1
    for (int col = 0; col < c; col++) // Change row to col
    {
        cout << vector[col] << endl; // Use endl instead of end1
    }
#pragma omp parallel // Move the parallel region outside the for loop
    {
#pragma omp for // Remove the inner parallel region
        for (int row = 0; row < r; row++)
        {
            out[row] = 0;
            for (int col = 0; col < c; col++) // Remove comma from for loop
            {
                out[row] += matrix[row][col] * vector[col];
            }
        }
    }
    cout << "Resultant Col-Vector" << endl; // Use endl instead of end1
    for (int row = 0; row < r; row++)
    {
```

```
      cout << "\nvector[" << row << "]:" << out[row] << endl; // Use endl instead of end1
   }
   return 0;
}
```

OUTPUT→



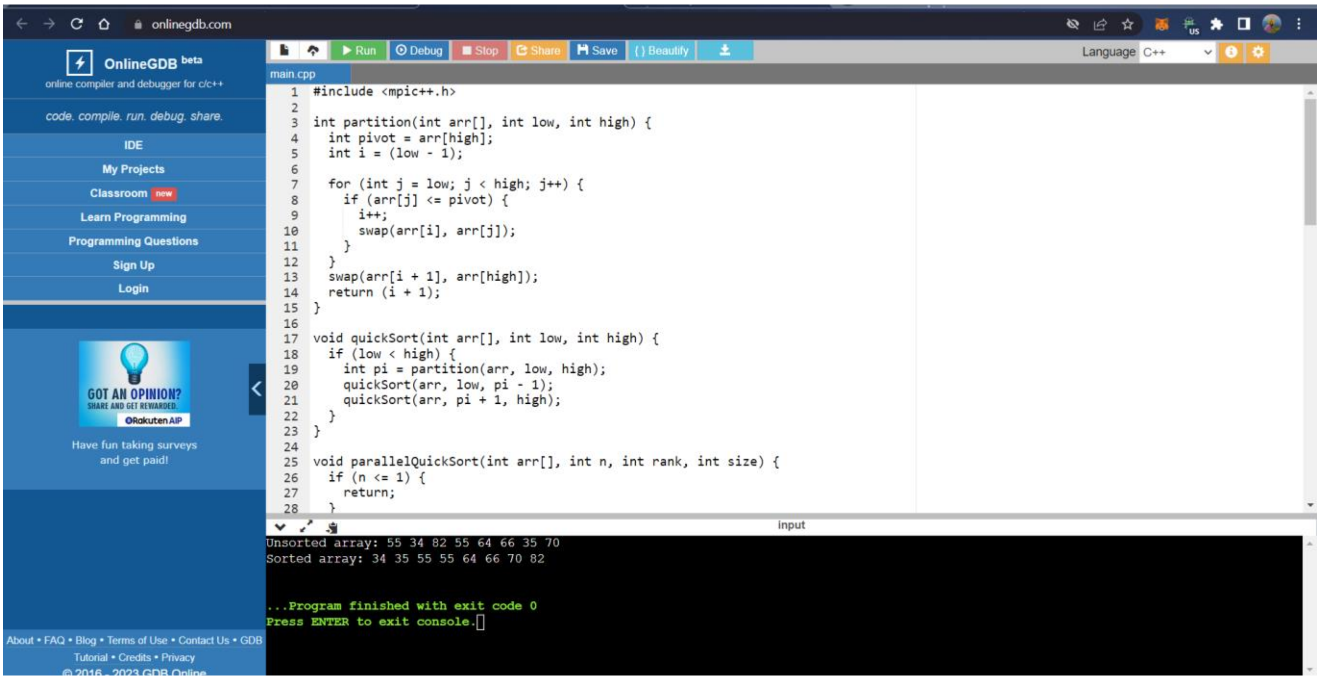
Mini Project

Title: Evaluate Performance enhancement of parallel Quick Sort Algorithm using MPI.

```
#include <mpi.h>

int partition(int arr[], int low, int high) {
  int pivot = arr[high];
  int i = (low - 1);

  for (int j = low; j < high; j++) {
    if (arr[j] <= pivot) {
      i++;
      swap(arr[i], arr[j]);
    }
  }
  swap(arr[i + 1], arr[high]);
  return (i + 1);
}

void quickSort(int arr[], int low, int high) {
  if (low < high) {
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}

void parallelQuickSort(int arr[], int n, int rank, int size) {
  if (n <= 1) {
    return;
  }

  int mid = n / 2;
  int pivot = arr[mid];

  // Scatter the array to the processes
  int local_arr[mid + 1];
  MPI_Scatter(arr, mid + 1, MPI_INT, local_arr, mid + 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Sort the local array
  quickSort(local_arr, 0, mid - 1);

  // Gather the sorted subarrays
  int global_arr[n];
  MPI_Gather(local_arr, mid + 1, MPI_INT, global_arr, mid + 1, MPI_INT, 0,
MPI_COMM_WORLD);
```

```
  // If this is the root process, print the sorted array
  if (rank == 0) {
    for (int i = 0; i < n; i++) {
      printf("%d ", global_arr[i]);
    }
    printf("\n");
  }
}

int main(int argc, char** argv) {
  int n;
  if (argc < 2) {
    printf("Usage: %s <array_size>\n", argv[0]);
    return 1;
  }

  n = atoi(argv[1]);

  // Initialize MPI
  MPI_Init(&argc, &argv);

  // Get the number of processes
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // Get the rank of the current process
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // Create an array
  int arr[n];
  for (int i = 0; i < n; i++) {
    arr[i] = rand() % 100;
  }

  // Sort the array in parallel
  parallelQuickSort(arr, n, rank, size);

  // Finalize MPI
  MPI_Finalize();

  return 0;
}
```

OUTPUT→


onlinegdb.com
Run
Debug
Stop
Share
Save
{ } Beautify
Language C++
OnlineGDB beta
online compiler and debugger for c/c++
code. compile. run. debug. share.
IDE
My Projects
Classroom new
Learn Programming
Programming Questions
Sign Up
Login
GOT AN OPINION?
SHARE AND GET REWARDED
Rakuten AIP
Have fun taking surveys and get paid!
About • FAQ • Blog • Terms of Use • Contact Us • GDB Tutorial • Credits • Privacy
© 2016 - 2023 GDB Online
main.cpp
#include <mpic++.h>
int partition(int arr[], int low, int high) {
int pivot = arr[high];
int i = (low - 1);
for (int j = low; j < high; j++) {
if (arr[j] <= pivot) {
i++;
swap(arr[i], arr[j]);
}
}
swap(arr[i + 1], arr[high]);
return (i + 1);
}
void quickSort(int arr[], int low, int high) {
if (low < high) {
int pi = partition(arr, low, high);
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
}
}
void parallelQuickSort(int arr[], int n, int rank, int size) {
if (n <= 1) {
return;
}
input
Unsorted array: 55 34 82 55 64 66 35 70
Sorted array: 34 35 55 55 64 66 70 82
...Program finished with exit code 0
Press ENTER to exit console.