

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→

← → ↻ 🌐 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

0 1 2 3 4 5 6 7 8 9

PREVIOUS: HELLO OPENMP

test.cpp

```

1 #include <iostream>
2 #include <queue>
3 #include <omp.h>
4
5 #define MAX_NODES 1000
6
7 using namespace std;
8
9 int visited[MAX_NODES];
10 int n_threads;
11
12 struct TreeNode {
13     int value;
14     vector<TreeNode*> children;
15 };
16
17 void bfs(TreeNode* root) {
18     queue<TreeNode*> q;
19     q.push(root);
20     visited[root->value] = 1;
21
22     while (!q.empty()) {
23         #pragma omp parallel num_threads(n_threads)
24         {
25             #pragma omp for
26             for (int i = 0; i < q.size(); i++) {
27                 TreeNode* node = q.front();
28                 q.pop();
29                 for (auto child : node->children) {
30                     if (!visited[child->value]) {
31                         visited[child->value] = 1;
32                         q.push(child);
33                     }
34                 }
35             }
36         }
37     }
38 }
    
```

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→

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

```
1 #include <iostream>
2 #include <vector>
3 #include <omp.h>
4
5 #define MAX_NODES 1000
6
7 using namespace std;
8
9 int result[MAX_NODES];
10 int n_threads;
11
12 struct TreeNode {
13     int value;
14     vector<TreeNode*> children;
15 };
16
17 void dfs(TreeNode* node, int depth) {
18     // Base case: leaf node
19     if (node->children.empty()) {
20         #pragma omp critical
21         {
22             result[depth] += node->value;
23         }
24     } else {
25         // Recursive case: internal node
26         #pragma omp parallel num_threads(n_threads)
27         {
28             #pragma omp single nowait
29             {
30                 for (auto child : node->children) {
31                     #pragma omp task
32                     {
33                         dfs(child, depth + 1);
34                     }
35                 }
36             }
37         }
38     }
```

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→

← → ↻ 📄 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

Bubble Sort execution time with 2 threads: 0.021894 seconds
Bubble Sort execution time without parallelism: 0.113968 seconds
Array is sorted

PREVIOUS: HELLO OPENMP

test.cpp

```

38 // Function to measure execution time
39 double measureExecutionTime(void (*sortFunction)(int[], int), int arr[], int n, int n_threads) {
40     double start_time, end_time;
41     omp_set_num_threads(n_threads);
42     start_time = omp_get_wtime();
43     sortFunction(arr, n);
44     end_time = omp_get_wtime();
45     return end_time - start_time;
46 }
47
48
49 int main() {
50     const int n = 10000;
51     int arr[n];
52     int n_threads = omp_get_max_threads();
53
54     // Generate random array
55     generateRandomArray(arr, n);
56
57     // Measure execution time of Bubble Sort
58     double bubbleSortSequentialTime = measureExecutionTime(bubbleSort, arr, n, 1);
59     double bubbleSortParallelTime = measureExecutionTime(bubbleSort, arr, n, n_threads);
60
61     // Print results
62     cout << "Bubble Sort execution time with " << n_threads << " threads: " << bubbleSortParallelTime << " seconds" << endl;
63     cout << "Bubble Sort execution time without parallelism: " << bubbleSortSequentialTime << " seconds" << endl;
64
65     // Check if array is sorted
66     if (isSorted(arr, n)) {
67         cout << "Array is sorted" << endl;
68     } else {
69         cout << "Array is not sorted" << endl;
70     }
71
72     return 0;
73 }
74

```

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);
    }
}

```

```

// 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

```

```

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

return 0;
}

```

OUTPUT→

The screenshot shows the codingame.com playground interface. The browser address bar displays 'codingame.com/playgrounds/54443/openmp/playground'. The page title is '2/2 Playground'. The main heading is 'OpenMP Playground'. Below the heading, it says 'Write any OpenMP code and run'. A file named 'test.cpp' is selected, and a green bar indicates 'SUCCESS!'. The 'Standard Output' section shows the following text: 'Merge Sort execution time with 2 threads: 0.00124097 seconds', 'Merge Sort execution time without parallelism: 0.00272489 seconds', and 'Array is sorted'. A 'PREVIOUS: HELLO OPENMP' button is visible. On the right, the 'test.cpp' code is displayed, showing a Merge Sort implementation with OpenMP parallelization and timing measurements.

test.cpp

```

99  }
100
101  int main() {
102      const int n = 10000;
103      int arr[n];
104      int n_threads = omp_get_max_threads();
105
106      // Generate random array
107      generateRandomArray(arr, n);
108
109      // Measure execution time of Merge Sort
110      double mergeSortSequentialTime = measureExecutionTime(mergeSort, arr, n, 1);
111      double mergeSortParallelTime = measureExecutionTime(mergeSort, arr, n, n_threads);
112
113      // Print results
114      cout << "Merge Sort execution time with " << n_threads << " threads: " << mergeSortParallelTime << " s\n";
115      cout << "Merge Sort execution time without parallelism: " << mergeSortSequentialTime << " seconds" << "\n";
116      // Verify that array is sorted
117
118
119      if (isSorted(arr, n)) {
120          cout << "Array is sorted" << endl;
121      } else {
122          cout << "Array is not sorted" << endl;
123      }
124
125      return 0;
126  }

```

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

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <ctime>
4 #include <omp.h>
5
6 using namespace std;
7
8 // function to generate random array
9 void generateRandomArray(int arr[], int n) {
10     srand(time(NULL));
11     for (int i = 0; i < n; i++) {
12         arr[i] = rand() % 100;
13     }
14 }
15
16 // function to find the minimum value in an array using parallel reduction
17 int findMin(int arr[], int n) {
18     int min_val = arr[0];
19     #pragma omp parallel for reduction(min:min_val)
20     for (int i = 1; i < n; i++) {
21         if (arr[i] < min_val) {
22             min_val = arr[i];
23         }
24     }
25     return min_val;
26 }
27
28 // function to find the maximum value in an array using parallel reduction
29 int findMax(int arr[], int n) {
30     int max_val = arr[0];
31     #pragma omp parallel for reduction(max:max_val)
32     for (int i = 1; i < n; i++) {
33         if (arr[i] > max_val) {
34             max_val = arr[i];
35         }
36     }
37     return max_val;
38 }
```

Experiment No:4

Addition of large vectors:-

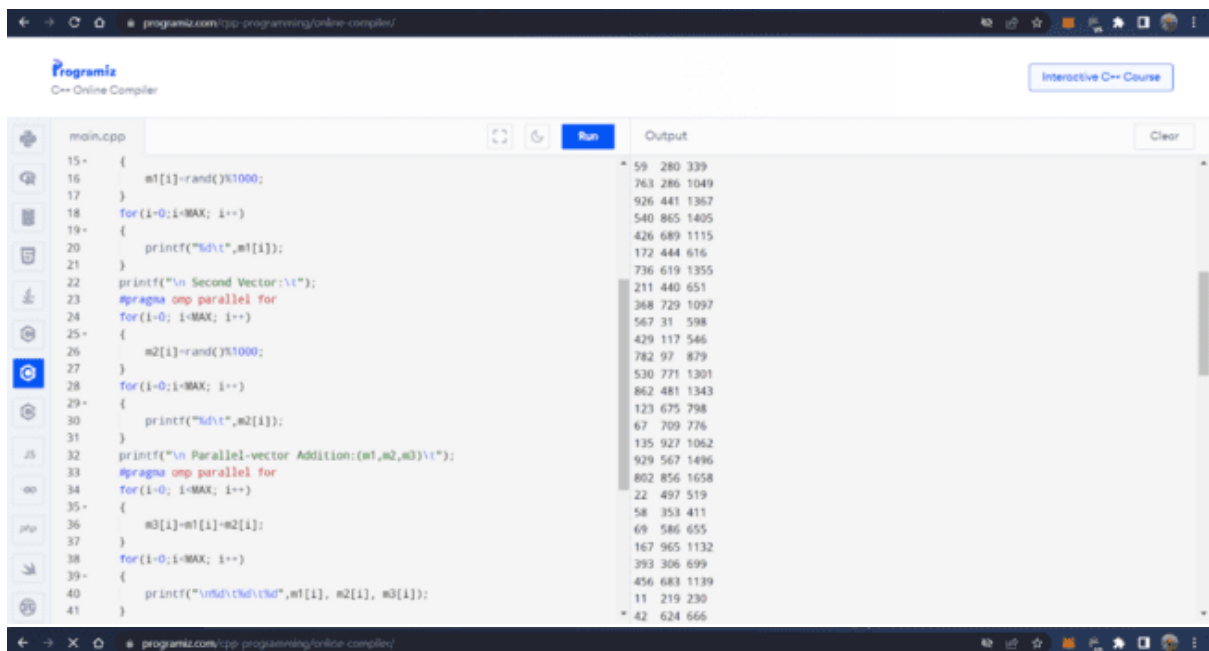
```
#include<stdio.h>
#include<iostream>
#include<cstdlib> /* CUDA Library */
#include<omp.h>

#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]);
    }
}
```

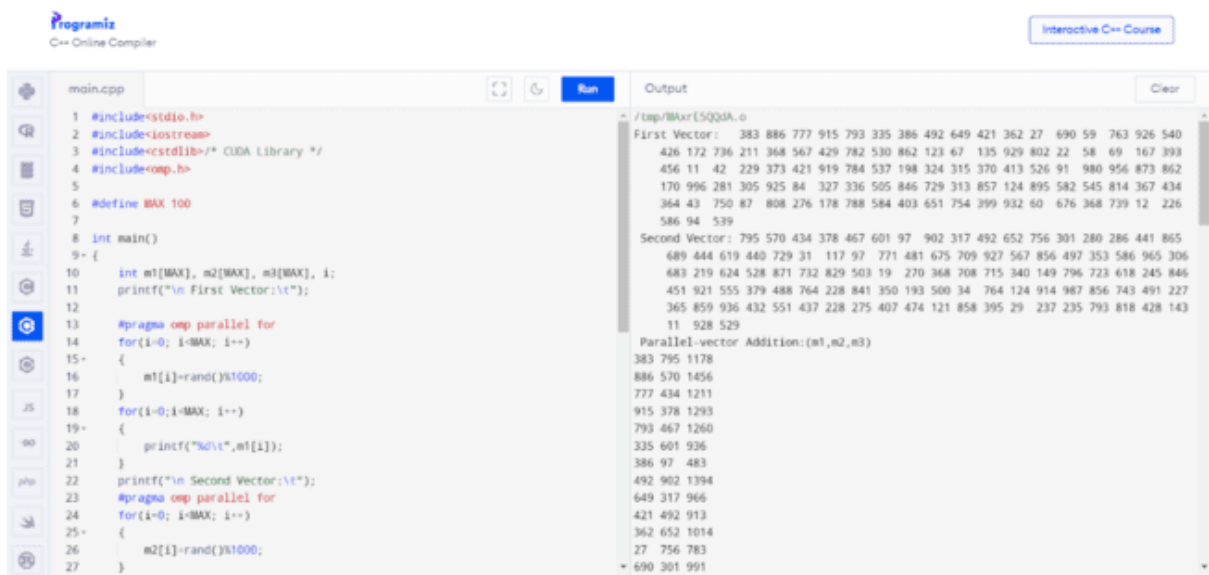
OUTPUT→



```
main.cpp
15+ {
16     m1[i]=rand()%1000;
17 }
18 for(i=0; i<MAX; i++)
19 {
20     printf("%d\t", m1[i]);
21 }
22 printf("\n Second Vector:\n");
23 #pragma omp parallel for
24 for(i=0; i<MAX; i++)
25 {
26     m2[i]=rand()%1000;
27 }
28 for(i=0; i<MAX; i++)
29 {
30     printf("%d\t", m2[i]);
31 }
32 printf("\n Parallel-vector Addition:(m1,m2,m3)\n");
33 #pragma omp parallel for
34 for(i=0; i<MAX; i++)
35 {
36     m3[i]=m1[i]+m2[i];
37 }
38 for(i=0; i<MAX; i++)
39 {
40     printf("\nd\t\t\t\t\t", m1[i], m2[i], m3[i]);
41 }
```

Output

```
59 280 339
763 286 1049
926 441 1367
540 865 1405
426 689 1115
172 444 616
736 619 1355
211 440 651
388 729 1097
567 31 598
429 117 546
782 97 879
530 771 1301
862 481 1343
123 675 798
67 709 776
135 927 1062
929 567 1496
802 856 1658
22 497 519
58 353 411
69 586 655
167 965 1132
393 306 699
456 683 1139
11 219 230
42 624 666
```



```
main.cpp
1 #include<stdio.h>
2 #include<iostream>
3 #include<cstdlib> /* CUBA Library */
4 #include<omp.h>
5
6 #define MAX 100
7
8 int main()
9 {
10     int m1[MAX], m2[MAX], m3[MAX], i;
11     printf("\n First Vector:\n");
12
13     #pragma omp parallel for
14     for(i=0; i<MAX; i++)
15     {
16         m1[i]=rand()%1000;
17     }
18     for(i=0; i<MAX; i++)
19     {
20         printf("%d\t", m1[i]);
21     }
22     printf("\n Second Vector:\n");
23     #pragma omp parallel for
24     for(i=0; i<MAX; i++)
25     {
26         m2[i]=rand()%1000;
27 }
```

Output

```
/tmp/BAKrESQqDA.o
First Vector: 383 886 777 915 793 335 386 492 649 421 362 27 690 59 763 926 540
426 172 736 211 368 567 429 782 530 862 123 67 135 929 802 22 58 69 167 393
456 11 42 229 373 421 919 784 537 198 324 315 370 413 526 91 980 956 873 862
170 996 281 305 925 84 327 336 505 846 729 313 857 124 895 582 545 814 367 434
364 43 750 87 808 276 178 788 584 403 651 754 399 932 60 676 368 739 12 226
586 94 539
Second Vector: 795 570 434 378 467 601 97 902 317 492 652 756 301 280 286 441 865
689 444 619 440 729 31 117 97 771 481 675 709 927 567 856 497 353 586 965 306
683 219 624 528 871 732 829 503 19 270 368 708 715 340 149 796 723 618 245 846
451 921 555 379 488 764 228 841 350 193 500 34 764 124 914 987 856 743 491 227
365 859 936 432 551 437 228 275 407 474 121 858 395 29 237 235 793 818 428 143
11 928 529
Parallel-vector Addition:(m1,m2,m3)
383 795 1178
886 570 1456
777 434 1211
915 378 1293
793 467 1260
335 601 936
386 97 483
492 902 1394
649 317 966
421 492 913
362 652 1014
27 756 783
690 301 991
```

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 endl
}
return 0;
}

```

OUTPUT→

The screenshot shows the Programiz C++ Online Compiler interface. The code editor on the left contains a C++ program that defines a 3x2 matrix and a 2x1 vector, multiplies them, and prints the result. The output window on the right shows the execution results.

```

main.cpp
1 #include <stdio.h>
2 #include <iostream>
3 #include <cstdlib>
4 #include <omp.h>
5
6 #define MAX 100
7
8 using namespace std; // Add this line to use cout and endl
9
10 int main()
11 {
12     int r = 3, c = 2;
13     int matrix[r][c], vector[c], out[r];
14
15     for (int row = 0; row < r; row++)
16     {
17         for (int col = 0; col < c; col++)
18         {
19             matrix[row][col] = 1;
20         }
21     }
22
23     cout << "Input Matrix" << endl; // Use endl instead of endl
24     for (int row = 0; row < r; row++)
25     {
26         for (int col = 0; col < c; col++)

```

Output

```

/tmp/MaxrESQq6A.o
Input Matrix
1 1
1 1
1 1
Input Col-Vector
2
2
Resultant Col-Vector
vector[0]:4
vector[1]:4
vector[2]:4

```

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→

The screenshot shows the OnlineGDB website interface. On the left is a sidebar with navigation links: OnlineGDB beta, code.compile.run.debug.share, IDE, My Projects, Classroom, Learn Programming, Programming Questions, Sign Up, and Login. Below these is a survey banner for 'GOT AN OPINION?' and a footer with links like About, FAQ, Blog, Terms of Use, Contact Us, GDB, Tutorial, Credits, Privacy, and Copyright 2016-2023 GDB Online.

The main area displays a C++ program in 'main.cpp' with the following code:

```
1 #include <mpic++.h>
2
3 int partition(int arr[], int low, int high) {
4     int pivot = arr[high];
5     int i = (low - 1);
6
7     for (int j = low; j < high; j++) {
8         if (arr[j] <= pivot) {
9             i++;
10            swap(arr[i], arr[j]);
11        }
12    }
13    swap(arr[i + 1], arr[high]);
14    return (i + 1);
15 }
16
17 void quickSort(int arr[], int low, int high) {
18     if (low < high) {
19         int pi = partition(arr, low, high);
20         quickSort(arr, low, pi - 1);
21         quickSort(arr, pi + 1, high);
22     }
23 }
24
25 void parallelQuickSort(int arr[], int n, int rank, int size) {
26     if (n <= 1) {
27         return;
28     }
29 }
```

Below the code editor is an 'input' field and a console output area. The output shows the execution results:

```
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.
```