LP-5 HPC
Asgn-1:
Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS.

Code

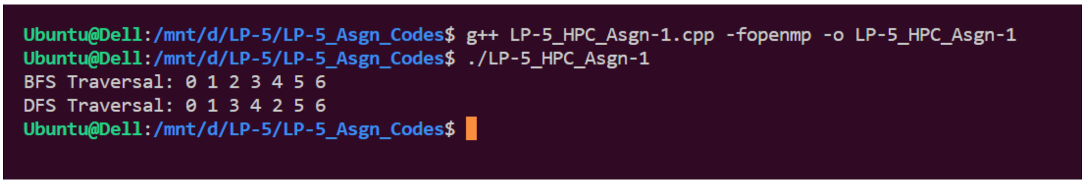
```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <omp.h>

using namespace std;

const int MAX_THREADS = 16; // Maximum number of threads to be used

// Graph class to represent an undirected graph
class Graph {
  int V; // number of vertices
  vector<int>* adj; // adjacency list
public:
  Graph(int V);
  void addEdge(int v, int w);
  void bfs(int start);
  void dfs(int start);
};

// Constructor to initialize a graph with V vertices
Graph::Graph(int V) {
  this->V = V;
  adj = new vector<int>[V];
}

// Function to add an edge between vertices v and w
void Graph::addEdge(int v, int w) {
  adj[v].push_back(w);
  adj[w].push_back(v);
}

// Breadth First Search algorithm
void Graph::bfs(int start) {
  // Mark all the vertices as not visited
  bool* visited = new bool[V];
  for(int i = 0; i < V; i++)
    visited[i] = false;

  // Create a queue for BFS
  queue<int> q;
```

```
    // Mark the current node as visited and enqueue it
    visited[start] = true;
    q.push(start);

    while(!q.empty()) {
        // Dequeue a vertex from queue and print it
        int s;
        #pragma omp critical
        {
            s = q.front();
            q.pop();
        }

        cout << s << " ";

        // Get all adjacent vertices of the dequeued vertex s.
        // If an adjacent has not been visited, then mark it visited and enqueue it
        #pragma omp parallel for num_threads(MAX_THREADS)
        for(int i = 0; i < adj[s].size(); i++) {
            int v = adj[s][i];
            if(!visited[v]) {
                #pragma omp critical
                {
                    visited[v] = true;
                    q.push(v);
                }
            }
        }
    }

    delete[] visited;
}
// Depth First Search algorithm
void Graph::dfs(int start) {
    // Mark all the vertices as not visited
    bool* visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a stack for DFS
    stack<int> s;

    // Push the current source node
    s.push(start);

    while(!s.empty()) {
        // Pop a vertex from stack and print it
        int v;
        #pragma omp critical
```

```
        {
          v = s.top();
          s.pop();
        }

        cout << v << " ";

        // Print if not visited and mark as visited
        if(!visited[v]) {
          #pragma omp critical
          {
            visited[v] = true;
          }
        }

        // Get all adjacent vertices of the popped vertex v.
        // If an adjacent has not been visited, then push it to the stack in reverse order
        #pragma omp parallel for num_threads(MAX_THREADS)
        for(int i = adj[v].size() - 1; i >= 0; i--) {
          int u = adj[v][i];
          if(!visited[u]) {
            #pragma omp critical
            {
              s.push(u);
            }
          }
        }
      }

      delete[] visited;
    }


    int main() {
      Graph g(7); // create a graph with 7 vertices
      g.addEdge(0, 1); // root node
      g.addEdge(0, 2);
      g.addEdge(1, 3);
      g.addEdge(1, 4);
      g.addEdge(2, 5);
       ddEdge(2, 6);


    // Call BFS and display complete traversal route
    cout << "BFS Traversal: ";
       fs(0);
    cout << endl;

    // Call DFS and display complete traversal route
    cout << "DFS Traversal: ";
```

```
g.dfs(0);
cout << endl;

return 0;
}
```

Output



```
Ubuntu@Dell:/mnt/d/LP-5/LP-5_Asgn_Codes$ g++ LP-5_HPC_Asgn-1.cpp -fopenmp -o LP-5_HPC_Asgn-1
Ubuntu@Dell:/mnt/d/LP-5/LP-5_Asgn_Codes$ ./LP-5_HPC_Asgn-1
BFS Traversal: 0 1 2 3 4 5 6
DFS Traversal: 0 1 3 4 2 5 6
Ubuntu@Dell:/mnt/d/LP-5/LP-5_Asgn_Codes$
```

LP-5 HPC
Asgn-2
Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

Code

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

using namespace std;
int n = 10;

// function to perform sequential bubble sort
void bubbleSort(int arr[], int n) {
   int i, j, temp;
   for (i = 0; i < n-1; i++) {
      for (j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
}

// function to perform parallel bubble sort using OpenMP
void parallelBubbleSort(int arr[], int n) {
   int i, j, temp;
   #pragma omp parallel for private(i, j, temp) num_threads(16)
   for (i = 0; i < n-1; i++) {
      for (j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
}

// function to merge two subarrays in ascending order
void merge(int arr[], int left, int mid, int right) {
   int i, j, k;
   int n1 = mid - left + 1;
   int n2 = right - mid;
   int L[n1], R[n2];
   for (i = 0; i < n1; i++) {
```

```
        L[i] = arr[left + i];
    }
    for (j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }
    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 mergeSort(int arr[], int left, int right, int n, bool isLastCall) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid, n, false);
        mergeSort(arr, mid+1, right, n, false);
        merge(arr, left, mid, right);
    }

}

// function to perform parallel merge sort using OpenMP
void parallelMergeSort(int arr[], int left, int right, int num_threads, int n) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        #pragma omp parallel sections num_threads(2)
        {
            #pragma omp section
            {
```

```
            parallelMergeSort(arr, left, mid, num_threads/2, n);
          }
          #pragma omp section
          {
            parallelMergeSort(arr, mid+1, right, num_threads/2, n);
          }
        }
        merge(arr, left, mid, right);
    }

}


int main() {

   int arr[n];

    cout << "Original Array: ";
    // initialize array with random values
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % n;
        cout << arr[i] <<" ";
    }

    // copy array for parallel sorting
    int arr_copy[n];
    for (int i = 0; i < n; i++) {
        arr_copy[i] = arr[i];
    }

    // measure time for sequential bubble sort
    double start_time = omp_get_wtime();
    bubbleSort(arr, n);
    double end_time = omp_get_wtime();
    double sequential_bubble_time = end_time - start_time;
    cout << "\n\nSequential Bubble Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] <<" ";
    }

    // measure time for parallel bubble sort
    start_time = omp_get_wtime();
    parallelBubbleSort(arr_copy, n);
    end_time = omp_get_wtime();
    double parallel_bubble_time = end_time - start_time;
    cout << "\nParallel Bubble Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] <<" ";
    }
    // output results for bubble sort
```

```
    cout << "\n\nBubble Sort Results:" << endl;
    cout << "Sequential Time: " << sequential_bubble_time << " seconds" << endl;
    cout << "Parallel Time: " << parallel_bubble_time << " seconds" << endl;

    cout << "\nOriginal Array: ";
    // reset array for merge sort
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % n;
        cout << arr[i] <<" ";
    }

    // copy array for parallel sorting
    for (int i = 0; i < n; i++) {
        arr_copy[i] = arr[i];
    }

    // measure time for sequential merge sort
    start_time = omp_get_wtime();
    mergeSort(arr, 0, n-1, n, true);
    end_time = omp_get_wtime();
    double sequential_merge_time = end_time - start_time;
    cout << "\n\nSequential Merge Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] <<" ";
    }
    // measure time for parallel merge sort
    start_time = omp_get_wtime();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp single
        {
            parallelMergeSort(arr_copy, 0, n-1, omp_get_num_threads(), n);
        }
    }
    end_time = omp_get_wtime();
    double parallel_merge_time = end_time - start_time;
    cout << "\nParallel Merge Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] <<" ";
    }
    // output results for merge sort
    cout <<endl << "\nMerge Sort Results:" << endl;
    cout << "Sequential Time: " << sequential_merge_time << " seconds" << endl;
    cout << "Parallel Time: " << parallel_merge_time << " seconds" << endl;

    return 0;
}
```

Output

```
● ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment2$ g++ LP-5_HPC_Asgn-2.cpp -fopenmp -o LP-5_HPC_Asgn-2
● ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment2$ ./LP-5_HPC_Asgn-2
  Original Array: 3 6 7 5 3 5 6 2 9 1

  Sequential Bubble Sorted Array: 1 2 3 3 5 5 6 6 7 9
  Parallel Bubble Sorted Array: 1 2 3 3 5 5 6 6 7 9

  Bubble Sort Results:
  Sequential Time: 5.7e-07 seconds
  Parallel Time: 0.00128508 seconds

  Original Array: 2 7 0 9 3 6 0 6 2 6

  Sequential Merge Sorted Array: 0 0 2 2 3 6 6 6 7 9
  Parallel Merge Sorted Array: 0 0 2 2 3 6 6 6 7 9

  Merge Sort Results:
  Sequential Time: 1.161e-06 seconds
  Parallel Time: 0.00148309 seconds
○ ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment2$
```

LP-5 HPC
Asgn-3
Implement Min, Max, Sum and Average operations using Parallel Reduction.

Code

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

using namespace std;

// function to generate an array of random integers
void generateArray(int arr[], int n) {
   for (int i = 0; i < n; i++) {
      arr[i] = rand();
   }
}

// function to perform parallel reduction to find the minimum value
int parallelMin(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 perform parallel reduction to find the maximum value
int parallelMax(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 perform parallel reduction to find the sum of values
int parallelSum(int arr[], int n) {
   int sum_val = 0;
   #pragma omp parallel for reduction(+:sum_val)
   for (int i = 0; i < n; i++) {
      sum_val += arr[i];
   }
```

```
    return sum_val;
}

// function to perform parallel reduction to find the average value
double parallelAverage(int arr[], int n) {
    double sum_val = 0.0;
    #pragma omp parallel for reduction(+:sum_val)
    for (int i = 0; i < n; i++) {
        sum_val += arr[i];
    }
    return sum_val / n;
}

int main() {
    int n = 10;
    int arr[n];
    generateArray(arr, n);

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << "\n\n";
    cout << "Minimum value: " << parallelMin(arr, n) << endl;
    cout << "Maximum value: " << parallelMax(arr, n) << endl;
    cout << "Sum of values: " << parallelSum(arr, n) << endl;
    cout << "Average value: " << parallelAverage(arr, n) << endl;
    return 0;
}
```

Output

```
abhi@Dell:/mnt/d/LP-5/LP-5_Asgn_Codes$ g++ LP-5_HPC_Asgn-3.cpp -fopenmp -o LP-5_HPC_Asgn-3
abhi@Dell:/mnt/d/LP-5/LP-5_Asgn_Codes$ ./LP-5_HPC_Asgn-3
83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21
19 84 37 98 24 15 70 13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64 43 50 87 8 76 78 88 84 3
51 54 99 32 60 76 68 39 12 26 86 94 39

Minimum value: 2
Maximum value: 99
Sum of values: 5184
Average value: 51.84
abhi@Dell:/mnt/d/LP-5/LP-5_Asgn_Codes$
```

LP-5 HPC
Asgn-4
Write a CUDA Program for:

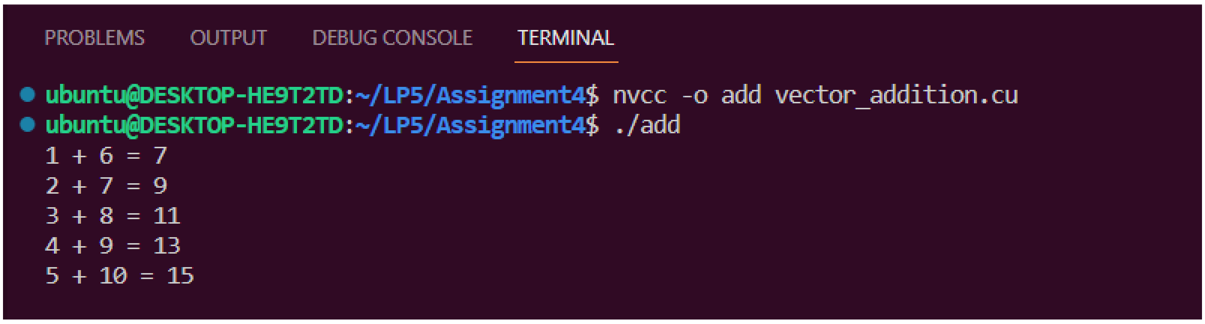
a) Addition of two large vectors
b) Matrix Multiplication using CUDA C

Code

```
#include <stdio.h>
#include <stdlib.h>

#define N 5

__global__void add(int *a, int *b, int *c) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i < N) {
      c[i] = a[i] + b[i];
   }
}

int main() {
   int a[N] = {1, 2, 3, 4, 5};
   int b[N] = {6, 7, 8, 9, 10};
   int c[N] = {0};

   int *dev_a, *dev_b, *dev_c;

   cudaMalloc((void **)&dev_a, N * sizeof(int));
   cudaMalloc((void **)&dev_b, N * sizeof(int));
   cudaMalloc((void **)&dev_c, N * sizeof(int));

   cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice);

   add<<<1, N>>>(dev_a, dev_b, dev_c);

   cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);

   for (int i = 0; i < N; i++) {
      //printf("%d ", c[i]);
      printf("%d + %d = %d\n", a[i], b[i], c[i]);
   }
   printf("\n");

   cudaFree(dev_a);
   cudaFree(dev_b);
   cudaFree(dev_c);

   return 0;
```

}

Output


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment4$ nvcc -o add vector_addition.cu
ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment4$ ./add
1 + 6 = 7
2 + 7 = 9
3 + 8 = 11
4 + 9 = 13
5 + 10 = 15

Code

```
#include <stdio.h>

#define N 3

__global__void matrixMultiplication(float *A, float *B, float *C, int n)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   if (i < n && j < n) {
      float sum = 0.0f;
      for (int k = 0; k < n; ++k) {
         sum += A[i * n + k] * B[k * n + j];
      }
      C[i * n + j] = sum;
   }
}

int main()
{
   float A[N][N] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   float B[N][N] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};
   float C[N][N] = {0};

   // Allocate device memory
   float *d_A, *d_B, *d_C;
   cudaMalloc(&d_A, N * N * sizeof(float));
   cudaMalloc(&d_B, N * N * sizeof(float));
   cudaMalloc(&d_C, N * N * sizeof(float));

   // Copy input matrices from host to device
   cudaMemcpy(d_A, A, N * N * sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, B, N * N * sizeof(float), cudaMemcpyHostToDevice);

   // Set the grid and block dimensions
   dim3 gridDim(ceil(N/16.0), ceil(N/16.0), 1);
   dim3 blockDim(16, 16, 1);

   // Launch the kernel
   matrixMultiplication<<<gridDim, blockDim>>>(d_A, d_B, d_C, N);

   // Copy result matrix from device to host
   cudaMemcpy(C, d_C, N * N * sizeof(float), cudaMemcpyDeviceToHost);

   // Print the result matrix
   printf("Result Matrix:\n");
   for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
```

```
            printf("%.1f ", C[i][j]);
        }
        printf("\n");
    }

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

Output

```
ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment4$ nvcc -o mul matrix_multiplication.cu
ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment4$ ./mul
Result Matrix:
30.0 24.0 18.0
84.0 69.0 54.0
138.0 114.0 90.0
```