Assignment 1

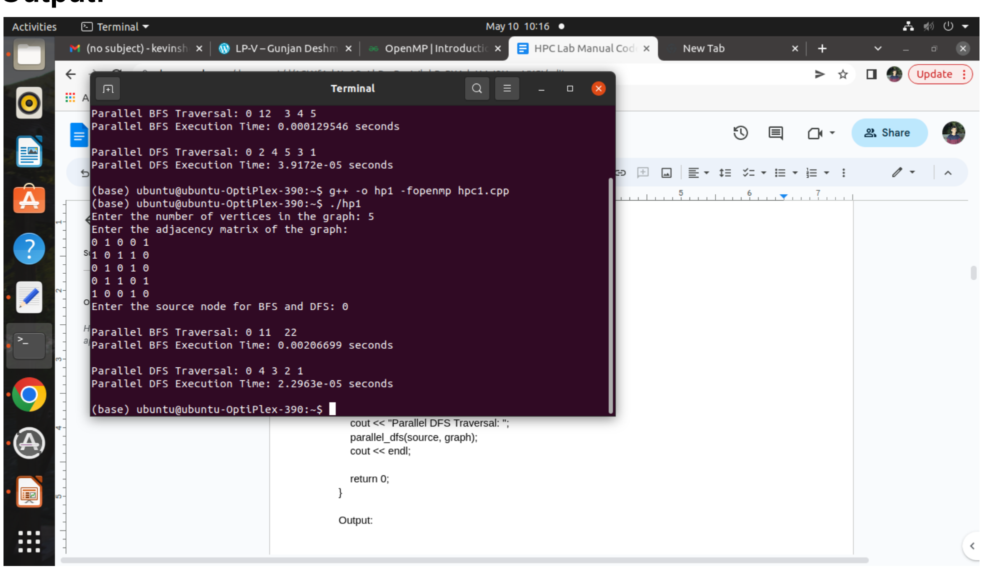
Program:

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

using namespace std;

void parallel_bfs(int source, vector<vector<int>>& graph)
{
    queue<int> q;
    int n = graph.size();
    vector<bool> visited(n, false);
    visited[source] = true;
    q.push(source);

    double start_time = omp_get_wtime();

    while (!q.empty())
    {
        int size = q.size();
#pragma omp parallel for schedule(dynamic)
        for (int i = 0; i < size; i++)
        {
            int current = q.front();
            q.pop();

            cout << current << " ";

            for (int j = 0; j < n; j++)
            {
                if (graph[current][j] == 1 && !visited[j])
                {
#pragma omp critical
                    {
                        visited[j] = true;
                        q.push(j);
                    }
                }
            }
```

```
        }
    }

    double end_time = omp_get_wtime();
    cout << "\nParallel BFS Execution Time: " << (end_time - start_time) << " seconds" << endl;
}

void parallel_dfs(int source, vector<vector<int>>& graph)
{
    stack<int> s;
    int n = graph.size();
    vector<bool> visited(n, false);
    visited[source] = true;
    s.push(source);

    double start_time = omp_get_wtime();

    while (!s.empty())
    {
        int current = s.top();
        s.pop();

        cout << current << " ";

#pragma omp parallel for schedule(dynamic)
        for (int i = 0; i < n; i++)
        {
            if (graph[current][i] == 1 && !visited[i])
            {
#pragma omp critical
                {
                    visited[i] = true;
                    s.push(i);
                }
            }
        }
    }

    double end_time = omp_get_wtime();
    cout << "\nParallel DFS Execution Time: " << (end_time - start_time) << " seconds" << endl;
}

int main()
{
```

```
    int n, source;
    cout << "Enter the number of vertices in the graph: ";
    cin >> n;

    vector<vector<int>> graph(n, vector<int>(n, 0));

    cout << "Enter the adjacency matrix of the graph:\n";
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cin >> graph[i][j];
        }
    }

    cout << "Enter the source node for BFS and DFS: ";
    cin >> source;

    cout << "\nParallel BFS Traversal: ";
    parallel_bfs(source, graph);
    cout << endl;

    cout << "Parallel DFS Traversal: ";
    parallel_dfs(source, graph);
    cout << endl;

    return 0;
}
```

Output:



Assignment 2

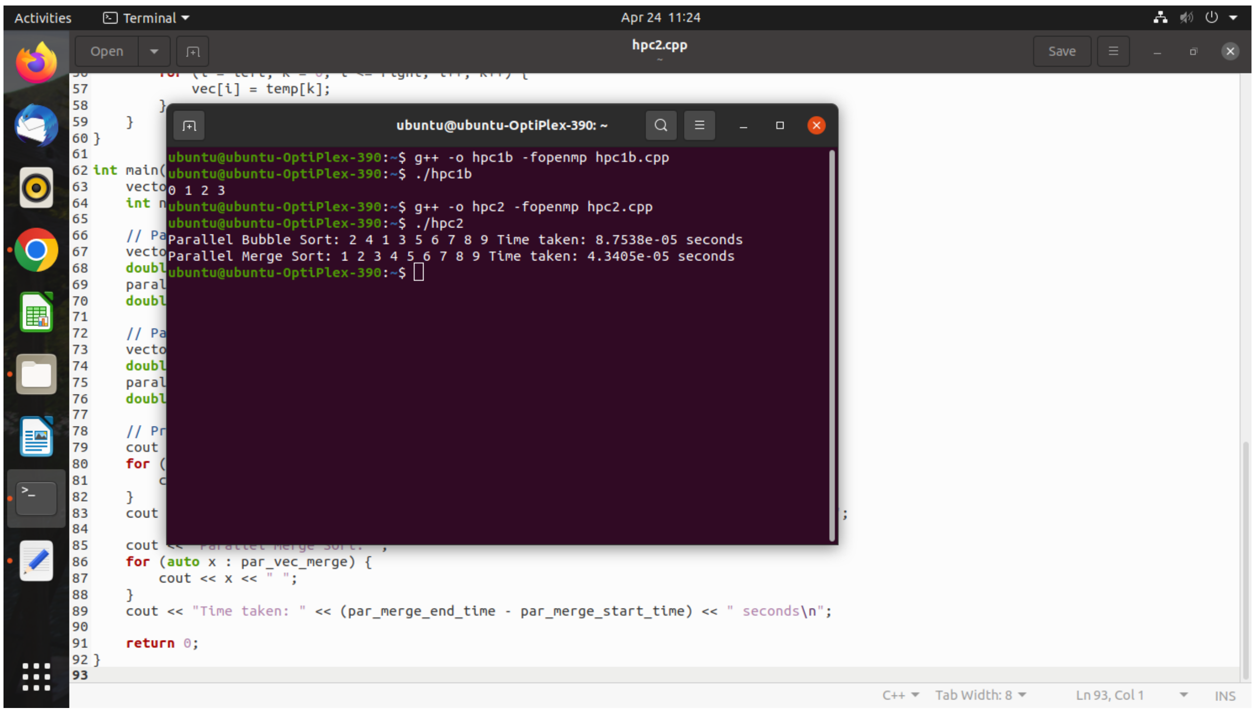
Program:

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

using namespace std;

// Parallel Bubble Sort
void parallel_bubble_sort(vector<int>& vec) {
    int n = vec.size();
    bool swapped = true;
    #pragma omp parallel default(none) shared(vec, n, swapped)
    {
        while (swapped) {
            swapped = false;
            #pragma omp for
            for (int i = 0; i < n-1; i++) {
                if (vec[i] > vec[i+1]) {
                    swap(vec[i], vec[i+1]);
                    swapped = true;
                }
            }
        }
    }
}

// Parallel Merge Sort
void parallel_merge_sort(vector<int>& vec, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        #pragma omp parallel sections
        {
            #pragma omp section
            {
                parallel_merge_sort(vec, left, mid);
            }
            #pragma omp section
            {
                parallel_merge_sort(vec, mid+1, right);
            }
        }
        vector<int> temp(right-left+1);
        int i = left, j = mid+1, k = 0;
```

```
        while (i <= mid && j <= right) {
            if (vec[i] < vec[j]) {
                temp[k++] = vec[i++];
            } else {
                temp[k++] = vec[j++];
            }
        }
        while (i <= mid) {
            temp[k++] = vec[i++];
        }
        while (j <= right) {
            temp[k++] = vec[j++];
        }
        for (i = left, k = 0; i <= right; i++, k++) {
            vec[i] = temp[k];
        }
    }
}

int main() {
    vector<int> vec = {9, 5, 7, 2, 8, 4, 1, 3, 6};
    int n = vec.size();

    // Parallel Bubble Sort
    vector<int> par_vec_bubble = vec;
    double par_bubble_start_time = omp_get_wtime();
    parallel_bubble_sort(par_vec_bubble);
    double par_bubble_end_time = omp_get_wtime();

    // Parallel Merge Sort
    vector<int> par_vec_merge = vec;
    double par_merge_start_time = omp_get_wtime();
    parallel_merge_sort(par_vec_merge, 0, n-1);
    double par_merge_end_time = omp_get_wtime();

    // Print results
    cout << "Parallel Bubble Sort: ";
    for (auto x : par_vec_bubble) {
        cout << x << " ";
    }
    cout << "Time taken: " << (par_bubble_end_time - par_bubble_start_time) << " seconds\n";

    cout << "Parallel Merge Sort: ";
    for (auto x : par_vec_merge) {
```

```
        cout << x << " ";
    }
    cout << "Time taken: " << (par_merge_end_time - par_merge_start_time) << " seconds\n";

    return 0;
}
```

Output:

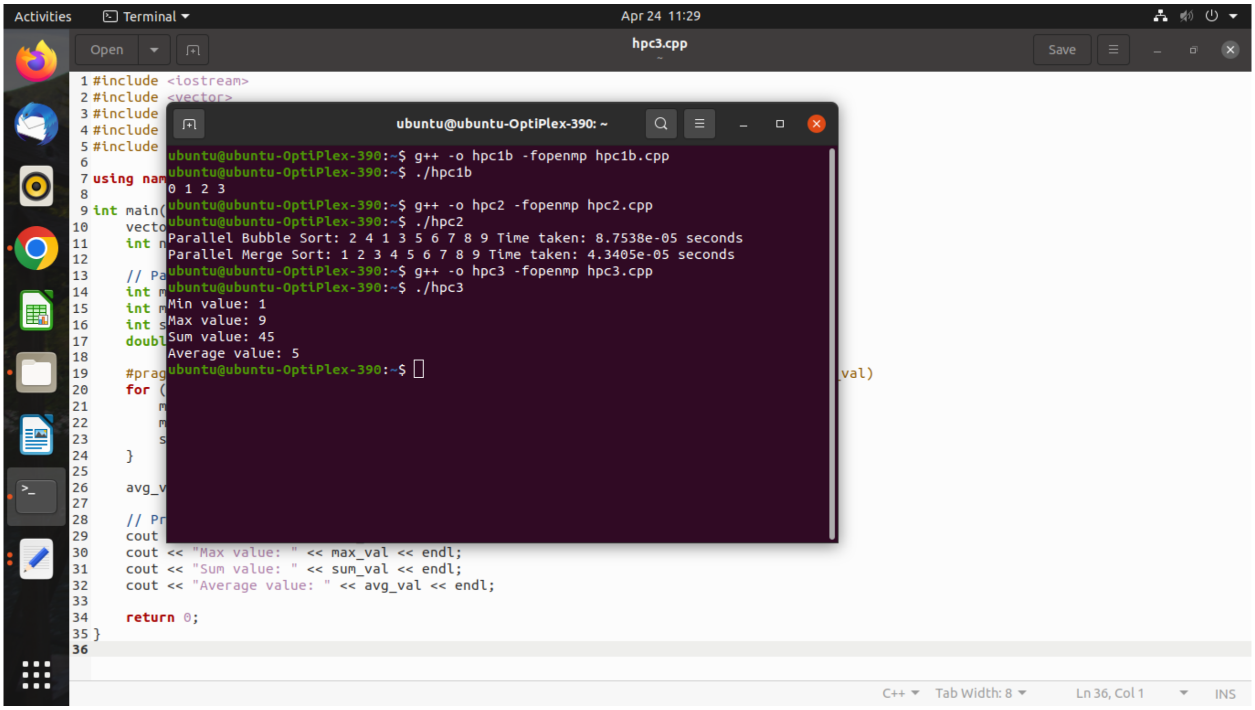


Assignment 3

Program:

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

using namespace std;

int main() {
    vector<int> vec = {9, 5, 7, 2, 8, 4, 1, 3, 6};
    int n = vec.size();

    // Parallel Reduction
    int min_val = vec[0];
    int max_val = vec[0];
    int sum_val = 0;
    double avg_val = 0;

    #pragma omp parallel for reduction(min:min_val) reduction(max:max_val)
reduction(+:sum_val)
    for (int i = 0; i < n; i++) {
        min_val = min(min_val, vec[i]);
        max_val = max(max_val, vec[i]);
        sum_val += vec[i];
    }

    avg_val = (double)sum_val / n;

    // Print results
    cout << "Min value: " << min_val << endl;
    cout << "Max value: " << max_val << endl;
    cout << "Sum value: " << sum_val << endl;
    cout << "Average value: " << avg_val << endl;

    return 0;
}
```

Output:


ubuntu@ubuntu-OptiPlex-390:~$ g++ -o hpc1b -fopenmp hpc1b.cpp
ubuntu@ubuntu-OptiPlex-390:~$ ./hpc1b
0 1 2 3
ubuntu@ubuntu-OptiPlex-390:~$ g++ -o hpc2 -fopenmp hpc2.cpp
ubuntu@ubuntu-OptiPlex-390:~$ ./hpc2
Parallel Bubble Sort: 2 4 1 3 5 6 7 8 9 Time taken: 8.7538e-05 seconds
Parallel Merge Sort: 1 2 3 4 5 6 7 8 9 Time taken: 4.3405e-05 seconds
ubuntu@ubuntu-OptiPlex-390:~$ g++ -o hpc3 -fopenmp hpc3.cpp
ubuntu@ubuntu-OptiPlex-390:~$ ./hpc3
Min value: 1
Max value: 9
Sum value: 45
Average value: 5
ubuntu@ubuntu-OptiPlex-390:~$

Assignment 4

Addition

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

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

int main() {
    int n = 1000000;
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size = n * sizeof(int);

    // Allocate memory on host
    a = (int*)malloc(size);
    b = (int*)malloc(size);
    c = (int*)malloc(size);

    // Initialize input arrays
    for (int i = 0; i < n; i++) {
        a[i] = i;
        b[i] = n - i;
    }

    // Allocate memory on device
    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);

    // Copy input data from host to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    add<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);
```

```
    // Copy output data from device to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Verify results
    for (int i = 0; i < n; i++) {
        if (c[i] != n) {
            printf("Error: c[%d] = %d\n", i, c[i]);
            break;
        }
    }

    // Free memory
    free(a);
    free(b);
    free(c);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

MAtrix Multiplication

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

#define N 1024

__global__ void matrixMul(int *a, int *b, int *c, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < n) {
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += a[row * n + i] * b[i * n + col];
        }
        c[row * n + col] = sum;
    }
}

int main() {
```

```
    int *h_a, *h_b, *h_c;
    int *d_a, *d_b, *d_c;
    int size = N * N * sizeof(int);

    // Allocate memory on host
    h_a = (int*)malloc(size);
    h_b = (int*)malloc(size);
    h_c = (int*)malloc(size);

    // Initialize input arrays
    for (int i = 0; i < N * N; i++) {
        h_a[i] = rand() % 10;
        h_b[i] = rand() % 10;
    }

    // Allocate memory on device
    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);

    // Copy input data from host to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Launch kernel
    dim3 threadsPerBlock(16, 16);
    dim3 blocksPerGrid((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (N + threadsPerBlock.y
- 1) / threadsPerBlock.y);
    matrixMul<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);

    // Copy output data from device to host
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    // Verify results
    for (int i = 0; i < N * N; i++) {
        if (h_c[i] != (h_a[i/N*N+i%N] * h_b[i%N*N+i/N])) {
            printf("Error: h_c[%d] = %d\n", i, h_c[i]);
            break;
        }
    }

    // Free memory
    free(h_a);
    free(h_b);
```

```
    free(h_c);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```