1. **Include necessary headers:**  
   #include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <omp.h>

* + iostream: For input/output operations.
  + vector: For dynamic array storage.
  + cstdlib and ctime: For random number generation.
  + omp.h: To use OpenMP for parallel processing.

1. **Merge Function:**  
   void merge(vector<int> &arr, int left, int mid, int right) { ... }
   * Merges two sorted halves of an array.
2. **Sequential Merge Sort:**  
   void mergeSort(vector<int> &arr, int left, int right) { ... }
   * Recursively divides the array and sorts it using the **merge** function.
3. **Parallel Merge Sort:**  
   #pragma omp parallel sections
   * Uses **OpenMP sections** to process left and right halves concurrently.
4. **Main function:**  
   int main() { ... }
   * Generates random numbers.
   * Measures execution time for sequential and parallel merge sort.
   * Uses omp\_get\_wtime() to measure time.

## **Compiling and Running the Program**

### **Compile:**

Use the following command to compile the program with OpenMP support:

g++ -fopenmp parallel\_merge\_sort.cpp -o parallel\_merge\_sort

### **Run:**

./parallel\_merge\_sort

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

// Function to merge two halves

void merge(vector<int> &arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

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

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

for (int i = 0; i < n2; i++)

R[i] = arr[mid + 1 + i];

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++;

}

}

// Sequential Merge Sort

void mergeSort(vector<int> &arr, int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

// Parallel Merge Sort

void parallelMergeSort(vector<int> &arr, int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

#pragma omp parallel sections

{

#pragma omp section

parallelMergeSort(arr, left, mid);

#pragma omp section

parallelMergeSort(arr, mid + 1, right);

}

merge(arr, left, mid, right);

}

}

int main() {

const int size = 100000;

vector<int> arr1(size), arr2(size);

srand(time(0));

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

arr1[i] = arr2[i] = rand() % 100000;

}

double start, end;

start = omp\_get\_wtime();

mergeSort(arr1, 0, size - 1);

end = omp\_get\_wtime();

cout << "Sequential Merge Sort Time: " << (end - start) << " seconds" << endl;

start = omp\_get\_wtime();

parallelMergeSort(arr2, 0, size - 1);

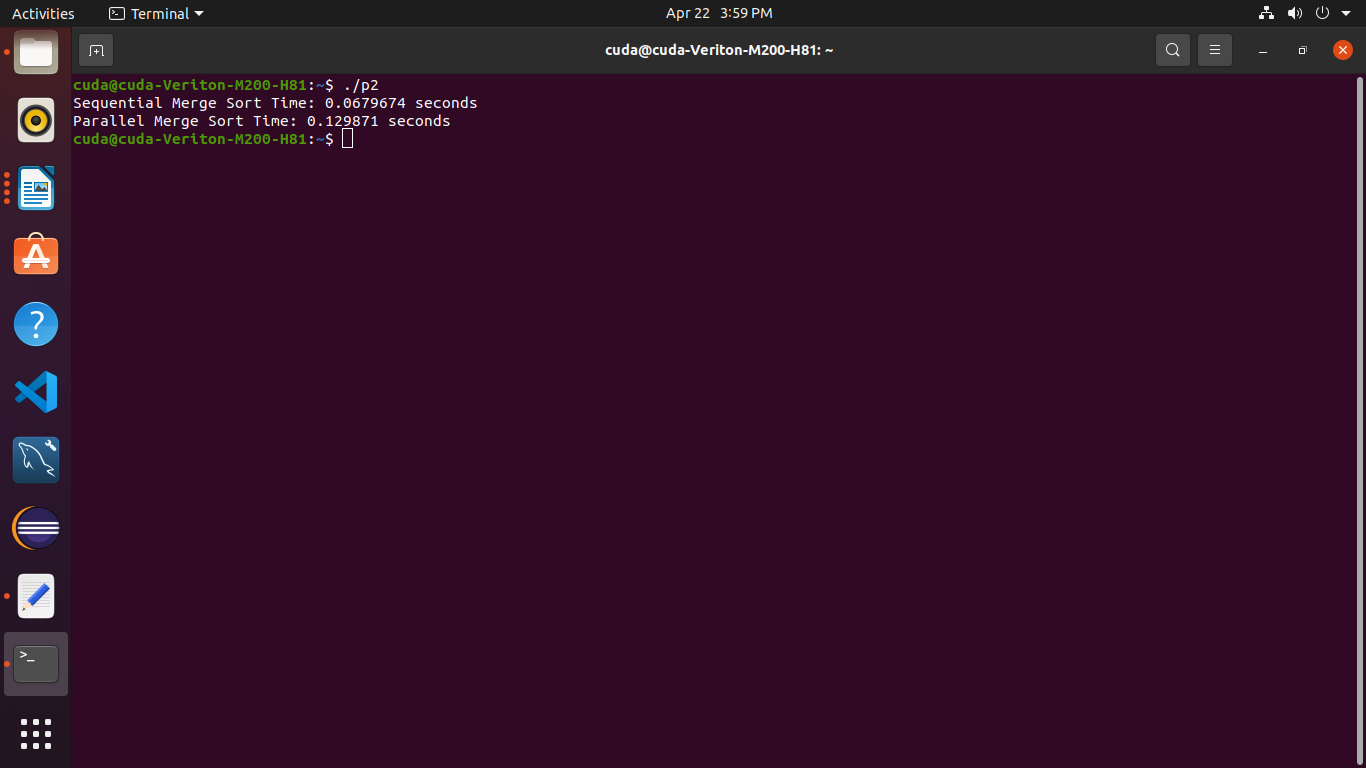
end = omp\_get\_wtime();

cout << "Parallel Merge Sort Time: " << (end - start) << " seconds" << endl;

return 0;

}

**Output:**

****