Parallel Bubble Sort using OpenMP:  
#include <iostream>

#include <omp.h>

using namespace std;

void parallel\_bubble\_sort(int \*arr, int n) {

int i, j, temp;

bool swapped;

#pragma omp parallel num\_threads(4) shared(arr, n) private(i, j, temp, swapped)

{

for (i = 0; i < n - 1; i++) {

swapped = false;

#pragma omp for

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;

swapped = true;

}

}

if (!swapped) break;

}

}

}

int main() {

int arr[] = {5, 2, 9, 1, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Before sorting: ";

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

cout << arr[i] << " ";

}

cout << endl;

parallel\_bubble\_sort(arr, n);

cout << "After sorting: ";

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

cout << arr[i] << " ";

}

cout << endl;

return 0;

}

Parallel Merge Sort using OpenMP:  
#include <iostream>

#include <omp.h>

using namespace std;

void merge(int \*arr, int l, int m, int r) {

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

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

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

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

i = 0;

j = 0;

k = l;

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++;

}

}

void sequential\_merge\_sort(int \*arr, int l, int r) {

if (l >= r) return;

int m = l + (r - l) / 2;

sequential\_merge\_sort(arr, l, m);

sequential\_merge\_sort(arr, m + 1, r);

merge(arr, l, m, r);

}

void parallel\_merge\_sort(int \*arr, int l, int r) {

if (l >= r) return;

int m = l + (r - l) / 2;

#pragma omp parallel num\_threads(2)

{

#pragma omp sections

{

#pragma omp section

parallel\_merge\_sort(arr, l, m);

#pragma omp section

parallel\_merge\_sort(arr, m + 1, r);

}

}

merge(arr, l, m, r);

}

int main() {

int arr[] = {5, 2, 9, 1, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Before sorting: ";

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

cout << arr[i] << " ";

}

cout << endl;

parallel\_merge\_sort(arr, 0, n - 1);

cout << "After sorting: ";

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

cout << arr[i] << " ";

}

cout << endl;

return 0;

}

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

#include <iostream>

#include <omp.h>

using namespace std;

void parallel\_min(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];

}

}

cout << "Minimum value in the array: " << min\_val << endl;

}

void parallel\_max(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];

}

}

cout << "Maximum value in the array: " << max\_val << endl;

}

void parallel\_sum(int \*arr, int n) {

int sum = 0;

#pragma omp parallel for reduction(+:sum)

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

sum += arr[i];

}

cout << "Sum of the array elements: " << sum << endl;

}

void parallel\_avg(int \*arr, int n) {

int sum = 0;

#pragma omp parallel for reduction(+:sum)

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

sum += arr[i];

}

double avg = static\_cast<double>(sum) / static\_cast<double>(n);

cout << "Average value of the array elements: " << avg << endl;

}

int main() {

int arr[] = {5, 2, 9, 1, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

parallel\_min(arr, n);

parallel\_max(arr, n);

parallel\_sum(arr, n);

parallel\_avg(arr, n);

return 0;

}

**CUDA program for addition of 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]);

}

}

**CUDA program for matrix multiplication**

#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;

}