Bubble sort🡪

#include <iostream>

#include <omp.h>

using namespace std;

void bubble(int \*, int);

void swap(int &, int &);

void bubble(int \*a, int n) {

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

int first = i % 2;

#pragma omp parallel for shared(a, first)

for (int j = first; j < n - 1; j += 2) {

if (a[j] > a[j + 1]) {

swap(a[j], a[j + 1]);

}

}

}

}

void swap(int &a, int &b) {

int tmp;

tmp = a;

a = b;

b = tmp;

}

int main() {

int \*a, n;

cout << "\n Enter total number: ";

cin >> n;

a = new int[n];

cout << " Enter number: ";

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

cin >> a[i];

}

bubble(a, n);

cout << " Sorted array: ";

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

cout << a[i] << " ";

}

delete[] a; // Free dynamically allocated memory

return 0;

}

Merge sort🡪

#include <iostream>

#include <vector>

#include <algorithm>

#include <thread>

using namespace std;

void merge(vector<int>& arr, int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

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

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

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

}

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

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

}

int 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 parallel\_merge\_sort(vector<int>& arr, int l, int r, int threads) {

if (l < r) {

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

if (threads > 1) {

thread left(parallel\_merge\_sort, ref(arr), l, m, threads / 2);

thread right(parallel\_merge\_sort, ref(arr), m + 1, r, threads / 2);

left.join();

right.join();

}

else {

parallel\_merge\_sort(arr, l, m, 1);

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

}

merge(arr, l, m, r);

}

}

int main() {

int m;

vector<int> arr;

cout << "\n--- Parallel Merge Sort ---" << endl;

cout << "Enter total number: ";

cin >> m;

cout << "Enter numbers: ";

int tmp;

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

cin >> tmp;

arr.push\_back(tmp);

}

int n = arr.size();

int threads = 4; // Number of threads to use

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

cout << "Sorted array: ";

for (auto x : arr) {

cout << x << " ";

}

cout << endl;

return 0;

}

Another merge sort

#include <iostream>

#include <omp.h>

using namespace std;

void merge(int arr[], int low, int mid, int high) {

// Create arrays of left and right partititons

int n1 = mid - low + 1;

int n2 = high - mid;

int left[n1];

int right[n2];

// Copy all left elements

for (int i = 0; i < n1; i++) left[i] = arr[low + i];

// Copy all right elements

for (int j = 0; j < n2; j++) right[j] = arr[mid + 1 + j];

// Compare and place elements

int i = 0, j = 0, k = low;

while (i < n1 && j < n2) {

if (left[i] <= right[j]){

arr[k] = left[i];

i++;

}

else{

arr[k] = right[j];

j++;

}

k++;

}

// If any elements are left out

while (i < n1) {

arr[k] = left[i];

i++;

k++;

}

while (j < n2) {

arr[k] = right[j];

j++;

k++;

}

}

void parallelMergeSort(int arr[], int low, int high) {

if (low < high) {

int mid = (low + high) / 2;

#pragma omp parallel sections

{

#pragma omp section

{

parallelMergeSort(arr, low, mid);

}

#pragma omp section

{

parallelMergeSort(arr, mid + 1, high);

}

}

merge(arr, low, mid, high);

}

}

void mergeSort(int arr[], int low, int high) {

if (low < high) {

int mid = (low + high) / 2;

mergeSort(arr, low, mid);

mergeSort(arr, mid + 1, high);

merge(arr, low, mid, high);

}

}

int main() {

int n = 10;

int arr[n];

double start\_time, end\_time;

// Create an array with numbers starting from n to 1.

for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

// Measure Sequential Time

start\_time = omp\_get\_wtime();

mergeSort(arr, 0, n - 1);

end\_time = omp\_get\_wtime();

cout << "Time taken by sequential algorithm: " << end\_time - start\_time << " seconds\n";

// Reset the array

for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

//Measure Parallel time

start\_time = omp\_get\_wtime();

parallelMergeSort(arr, 0, n - 1);

end\_time = omp\_get\_wtime();

cout << "Time taken by parallel algorithm: " << end\_time - start\_time << " seconds";

return 0;

}