**PP LAB WEEK-4**

# DSE VI-A2 Divansh Prasad 210968140

1) Write a parallel program using OpenMP to implement the Selection sort algorithm. Compute the efficiency and plot the speed up for varying input size and thread number.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

#define MAX\_VALUE 100

void generateArray(int arr[], int size) {

srand(time(NULL));

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

arr[i] = rand() % MAX\_VALUE;

}

}

void selectionSort(int arr[], int n) {

int i, j, min\_idx;

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

min\_idx = i;

for (j = i + 1; j < n; j++) {

if (arr[j] < arr[min\_idx]) {

min\_idx = j;

}

}

if (min\_idx != i) {

int temp = arr[i];

arr[i] = arr[min\_idx];

arr[min\_idx] = temp;

}

}

}

double sequentialSort(int arr[], int size) {

clock\_t start, end;

double cpu\_time\_used;

start = clock();

selectionSort(arr, size);

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

return cpu\_time\_used;

}

double parallelSort(int arr[], int size, int num\_threads) {

clock\_t start, end;

double cpu\_time\_used;

start = clock();

#pragma omp parallel for num\_threads(num\_threads)

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

selectionSort(arr, size);

}

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

return cpu\_time\_used;

}

int main() {

printf("Array Size\tThreads\tSequential Time (s)\tParallel Time (s)\tSpeedup\t\tEfficiency\n");

for (int size = 200; size <= 1000; size += 200) {

int arr[size];

generateArray(arr, size);

for (int num\_threads = 2; num\_threads <= 8; num\_threads += 2) {

double sequential\_time = sequentialSort(arr, size);

double parallel\_time = parallelSort(arr, size, num\_threads);

double speedup = sequential\_time / parallel\_time;

double efficiency = speedup / num\_threads;

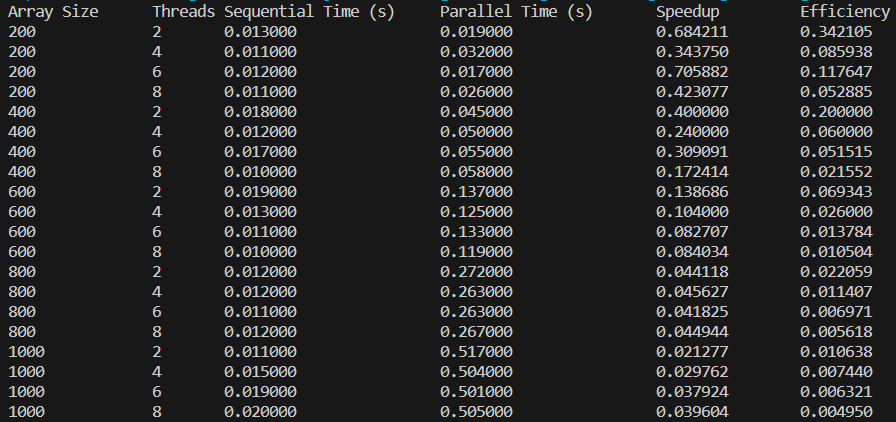
printf("%d\t\t%d\t%.6f\t\t%.6f\t\t%.6f\t%.6f\n", size, num\_threads, sequential\_time, parallel\_time, speedup, efficiency);

}

}

return 0;

}



2) Write a parallel program using openMP to implement the following: Take an array of input size m. Divide the array into two parts and sort the first half using insertion sort and second half using quick sort. Use two threads to perform these tasks. Use merge sort to combine the results of these two sorted arrays.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

void generateArray(int arr[], int size) {

srand(time(NULL));

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

arr[i] = rand() % 100;

}

}

void insertionSortSequential(int arr[], int size) {

int i, key, j;

for (i = 1; i < size; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

void insertionSortParallel(int arr[], int size, int num\_threads) {

#pragma omp parallel for num\_threads(num\_threads)

for (int i = 1; i < size; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

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

if (low < high) {

int pivot = arr[low];

int i = low;

int j = high;

while (i < j) {

while (arr[i] <= pivot && i <= high - 1) {

i++;

}

while (arr[j] > pivot && j >= low + 1) {

j--;

}

if (i < j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[low];

arr[low] = arr[j];

arr[j] = temp;

quickSortSequential(arr, low, j - 1);

quickSortSequential(arr, j + 1, high);

}

}

void quickSortParallel(int arr[], int low, int high, int num\_threads) {

if (low < high) {

int pivot = arr[low];

int i = low;

int j = high;

while (i < j) {

while (arr[i] <= pivot && i <= high - 1) {

i++;

}

while (arr[j] > pivot && j >= low + 1) {

j--;

}

if (i < j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[low];

arr[low] = arr[j];

arr[j] = temp;

#pragma omp parallel sections num\_threads(num\_threads)

{

#pragma omp section

quickSortParallel(arr, low, j - 1, num\_threads);

#pragma omp section

quickSortParallel(arr, j + 1, high, num\_threads);

}

}

}

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 sequentialSort(int arr[], int size) {

int mid = size / 2;

insertionSortSequential(arr, size);

quickSortSequential(arr, mid, size - 1);

merge(arr, 0, mid - 1, size - 1);

}

void parallelSort(int arr[], int size, int num\_threads) {

int mid = size / 2;

#pragma omp parallel sections

{

#pragma omp section

{

insertionSortParallel(arr, size, num\_threads);

}

#pragma omp section

{

quickSortParallel(arr, mid, size - 1, num\_threads);

}

}

merge(arr, 0, mid - 1, size - 1);

}

int main() {

clock\_t start, end;

double cpu\_time\_used\_sequential = 0;

double cpu\_time\_used\_parallel = 0;

int num\_threads = 1;

printf("Array Size\tThreads\tSequential Time (s)\tParallel Time (s)\tSpeedup\t\tEfficiency\n");

for (int size = 200; size <= 1000; size += 200) {

for (num\_threads = 2; num\_threads <= 8; num\_threads += 2) {

int arr[size];

generateArray(arr, size);

start = clock();

sequentialSort(arr, size);

end = clock();

cpu\_time\_used\_sequential = ((double)(end - start)) / CLOCKS\_PER\_SEC;

start = clock();

parallelSort(arr, size, num\_threads);

end = clock();

cpu\_time\_used\_parallel = ((double)(end - start)) / CLOCKS\_PER\_SEC;

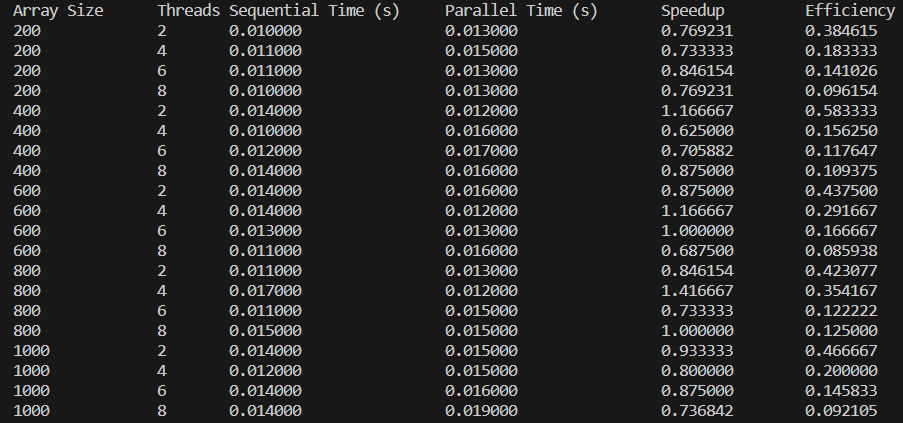
printf("%d\t\t%d\t%.6f\t\t%.6f\t\t%.6f\t%.6f\n", size, num\_threads, cpu\_time\_used\_sequential, cpu\_time\_used\_parallel, cpu\_time\_used\_sequential / cpu\_time\_used\_parallel, (cpu\_time\_used\_sequential / cpu\_time\_used\_parallel) / num\_threads);

}

}

return 0;

}



3) Write a parallel program using OpenMP to implement sequential search algorithm. Compute the efficiency and plot the speed up for varying input size and thread number.

#include <stdio.h>

#include <omp.h>

#include <time.h>

#include <windows.h>

#include <stdbool.h>

bool sequentialSearch(int element, int array[], int n) {

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

if (array[i] == element) {

return true;

}

}

return false;

}

bool parallelSearch(int element, int array[], int n, int num\_threads) {

bool found = false;

#pragma omp parallel for num\_threads(num\_threads) shared(found)

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

if (array[i] == element) {

found = true;

}

}

return found;

}

int main() {

printf("Array Size\tThreads\t Sequential Time (s)\t Parallel Time (s)\tSpeedup\t\tEfficiency\n");

for (int size = 200; size <= 800; size += 200) {

int arr[size];

for (int num\_threads = 2; num\_threads <= 8; num\_threads += 2) {

double sequential\_time = 0;

double parallel\_time = 0;

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

arr[i] = rand() % 100;

}

int element\_to\_find = arr[rand() % 100];

clock\_t start = clock();

sequentialSearch(element\_to\_find, arr, size);

clock\_t end = clock();

sequential\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

start = clock();

parallelSearch(element\_to\_find, arr, size, num\_threads);

end = clock();

parallel\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

double speedup = sequential\_time / parallel\_time;

double efficiency = speedup / num\_threads;

printf("%d\t\t%d\t%.6f\t\t%.6f\t\t%.6f\t%.6f\n", size,num\_threads, sequential\_time, parallel\_time, speedup, efficiency);

}

}

return 0;

}

