Distributed and Parallel Computing Lab

CS461 Lab9

Name: Dipean Dasgupta **ID:202151188**

Task: Implementation of Parallel quicksort using OpenMP Libraries and comparison with normal quicksort.

IDE: VS Code

Program Language: C **Library used: OpenMP**

Code:

```c
#include <stdio.h>
#include <windows.h>
#include <omp.h>

//Array Partition and pivot index return
int partition(int arr[], int left, int right) {
    int pivot = arr[left];
    int l = left, r = right;

    while (l < r) {
        while (arr[r] > pivot) {
            r--;
        }
        while (l < r && arr[l] <= pivot) {
            l++;
        }
        if (l < r) {
            int temp = arr[l];
            arr[l] = arr[r];
            arr[r] = temp;
        }
    }
    int temp = arr[left];
    arr[left] = arr[r];
    arr[r] = temp;

    return r;
}

// Recursive function
void quicksort(int arr[], int left, int right) {
    if (left < right) {
```

```
        int pivotIndex = partition(arr, left, right);

        #pragma omp parallel sections
        {
            #pragma omp section
            quicksort(arr, left, pivotIndex - 1);
            #pragma omp section
            quicksort(arr, pivotIndex + 1, right);
        }
    }
}

// Regular quicksort function (without OpenMP)
void regular_quicksort(int arr[], int left, int right) {
    if (left < right) {
        int pivotIndex = partition(arr, left, right);
        regular_quicksort(arr, left, pivotIndex - 1);
        regular_quicksort(arr, pivotIndex + 1, right);
    }
}

int main() {
    int arr[] = {79, 17, 14, 65, 89, 4, 95, 22, 63, 11,146, 19, 81, 68, 39, 3,
52, 109, 77, 124, 112,191,176,13,26,37,89};
    int size = sizeof(arr) / sizeof(arr[0]);
    int arr_copy[size];
    for (int i = 0; i < size; i++) {
        arr_copy[i] = arr[i];
    }

    // Variables for high-resolution timing
    LARGE_INTEGER start, end, frequency;

    // Measure time for parallel quicksort
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&start);
    #pragma omp parallel
    {
        #pragma omp single
        quicksort(arr, 0, size - 1);
    }
    QueryPerformanceCounter(&end);
    double time_taken_parallel = (double)(end.QuadPart - start.QuadPart) *
1000000000.0 / frequency.QuadPart;
```

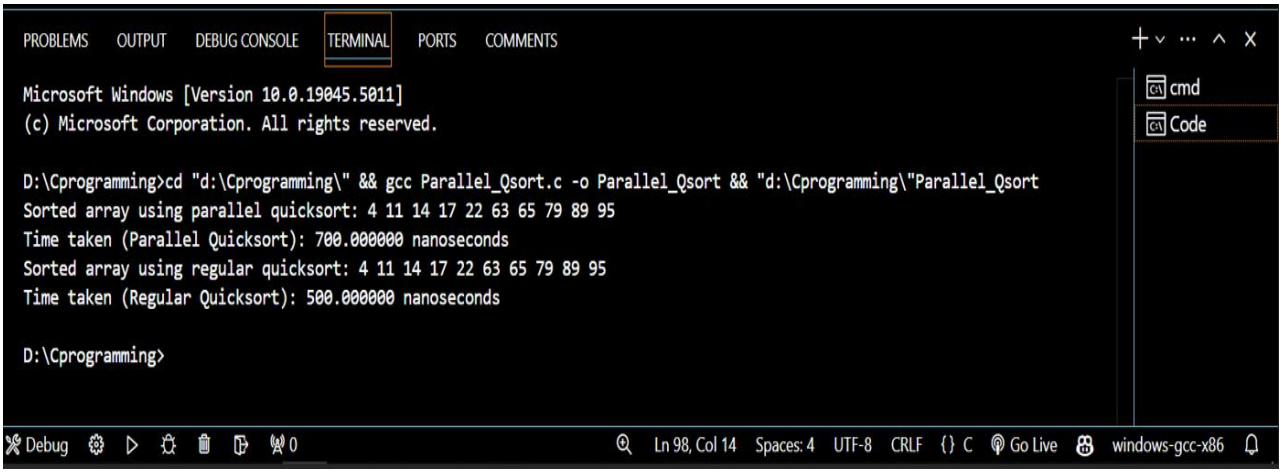
```
    // Printing the sorted array
    printf("Sorted array using parallel quicksort: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\nTime taken (Parallel Quicksort): %f nanoseconds\n",
time_taken_parallel);

    // Measure time for regular quicksort
    QueryPerformanceCounter(&start);
    regular_quicksort(arr_copy, 0, size - 1);
    QueryPerformanceCounter(&end);
    double time_taken_regular = (double)(end.QuadPart - start.QuadPart) *
1000000000.0 / frequency.QuadPart;

    // Printing the sorted array
    printf("Sorted array using regular quicksort: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr_copy[i]);
    }
    printf("\nTime taken (Regular Quicksort): %f nanoseconds\n",
time_taken_regular);
    return 0;
}
```

OUTPUT:

Case 1: Smaller array



Here, execution time: parallel quicksort>regular quicksort

Regular quicksort sorted the array faster.

Case2: Mid sized Array

```
D:\Cprogramming>cd "d:\Cprogramming\" && gcc Parallel_Qsort.c -o Parallel_Qsort && "d:\Cprogramming\"Parallel_Qsort
Sorted array using parallel quicksort: 3 4 11 14 17 19 22 39 52 63 65 68 77 79 81 89 95 109 124 146
Time taken (Parallel Quicksort): 800.000000 nanoseconds
Sorted array using regular quicksort: 3 4 11 14 17 19 22 39 52 63 65 68 77 79 81 89 95 109 124 146
Time taken (Regular Quicksort): 800.000000 nanoseconds
```

Here, still execution time: parallel quicksort>= regular quicksort

Both took equal time to sort with a mid sized array.

Case 3: Larger Array

```
D:\Cprogramming>cd "d:\Cprogramming\" && gcc Parallel_Qsort.c -o Parallel_Qsort && "d:\Cprogramming\"Parallel_Qsort
Sorted array using parallel quicksort: 3 4 11 13 14 17 19 22 26 37 39 52 63 65 68 77 79 81 89 89 95 109 112 124 146 176 191

Time taken (Parallel Quicksort): 1100.000000 nanoseconds
Sorted array using regular quicksort: 3 4 11 13 14 17 19 22 26 37 39 52 63 65 68 77 79 81 89 89 95 109 112 124 146 176 191
Time taken (Regular Quicksort): 1300.000000 nanoseconds
```

Here, execution time: Regular quicksort> Parallel quicksort

For larger arrays, parallel quicksort executes faster than normal quicksort.

So, we can conclude that parallel quicksort will be handy while sorting large or very large arrays because it divides the array into subproblems and handles these concurrently. On the other hand, normal quicksort will complete the sorting efficiently if the array size is small.

For Small Array: normal quicksort

For Large Array: Parallel quicksort