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);
```

```c
            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;
}
```

Array Size	Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency
200	2	0.013000	0.019000	0.684211	0.342105
200	4	0.011000	0.032000	0.343750	0.085938
200	6	0.012000	0.017000	0.705882	0.117647
200	8	0.011000	0.026000	0.423077	0.052885
400	2	0.018000	0.045000	0.400000	0.200000
400	4	0.012000	0.050000	0.240000	0.060000
400	6	0.017000	0.055000	0.309091	0.051515
400	8	0.010000	0.058000	0.172414	0.021552
600	2	0.019000	0.137000	0.138686	0.069343
600	4	0.013000	0.125000	0.104000	0.026000
600	6	0.011000	0.133000	0.082707	0.013784
600	8	0.010000	0.119000	0.084034	0.010504
800	2	0.012000	0.272000	0.044118	0.022059
800	4	0.012000	0.263000	0.045627	0.011407
800	6	0.011000	0.263000	0.041825	0.006971
800	8	0.012000	0.267000	0.044944	0.005618
1000	2	0.011000	0.517000	0.021277	0.010638
1000	4	0.015000	0.504000	0.029762	0.007440
1000	6	0.019000	0.501000	0.037924	0.006321
1000	8	0.020000	0.505000	0.039604	0.004950

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;
}
```

Array Size	Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency
200	2	0.010000	0.013000	0.769231	0.384615
200	4	0.011000	0.015000	0.733333	0.183333
200	6	0.011000	0.013000	0.846154	0.141026
200	8	0.010000	0.013000	0.769231	0.096154
400	2	0.014000	0.012000	1.166667	0.583333
400	4	0.010000	0.016000	0.625000	0.156250
400	6	0.012000	0.017000	0.705882	0.117647
400	8	0.014000	0.016000	0.875000	0.109375
600	2	0.014000	0.016000	0.875000	0.437500
600	4	0.014000	0.012000	1.166667	0.291667
600	6	0.013000	0.013000	1.000000	0.166667
600	8	0.011000	0.016000	0.687500	0.085938
800	2	0.011000	0.013000	0.846154	0.423077
800	4	0.017000	0.012000	1.416667	0.354167
800	6	0.011000	0.015000	0.733333	0.122222
800	8	0.015000	0.015000	1.000000	0.125000
1000	2	0.014000	0.015000	0.933333	0.466667
1000	4	0.012000	0.015000	0.800000	0.200000
1000	6	0.014000	0.016000	0.875000	0.145833
1000	8	0.014000	0.019000	0.736842	0.092105

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++) {
```

```c
                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;
}
```

Array Size	Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency
200	2	0.010000	0.016000	0.625000	0.312500
200	4	0.011000	0.019000	0.578947	0.144737
200	6	0.011000	0.015000	0.733333	0.122222
200	8	0.014000	0.013000	1.076923	0.134615
400	2	0.011000	0.016000	0.687500	0.343750
400	4	0.011000	0.019000	0.578947	0.144737
400	6	0.014000	0.016000	0.875000	0.145833
400	8	0.010000	0.013000	0.769231	0.096154
600	2	0.011000	0.013000	0.846154	0.423077
600	4	0.010000	0.013000	0.769231	0.192308
600	6	0.011000	0.012000	0.916667	0.152778
600	8	0.011000	0.012000	0.916667	0.114583
800	2	0.014000	0.013000	1.076923	0.538462
800	4	0.011000	0.012000	0.916667	0.229167
800	6	0.015000	0.012000	1.250000	0.208333
800	8	0.011000	0.012000	0.916667	0.114583
1000	2	0.010000	0.013000	0.769231	0.384615
1000	4	0.011000	0.012000	0.916667	0.229167
1000	6	0.011000	0.012000	0.916667	0.152778
1000	8	0.015000	0.012000	1.250000	0.156250