PP LAB WEEK-5

DSE VI-A2 Divansh Prasad 210968140

1) Write a parallel program using OpenMP to perform vector addition, subtraction, multiplication. Demonstrate task level parallelism. Analyze the speedup and efficiency of the parallelized code.

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

void generateRandomVector(int *vector, int size) {
    srand(time(NULL));
    for (int i = 0; i < size; i++) {
        vector[i] = rand() % 100;
    }
}

void VectorAddition(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}

void VectorSubtraction(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] - b[i];
    }
}

void VectorMultiplication(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] * b[i];
    }
}

int main() {
```

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

    for (int size = 200; size <= 800; size += 200) {
        int v1[size], v2[size], add[size], sub[size], mult[size];

        generateRandomVector(v1, size);
        generateRandomVector(v2, size);

        for (int num_threads = 2; num_threads <= 8; num_threads += 2) {
            double sequential_time = 0, parallel_time = 0;

            // Sequential operations
            clock_t start = clock();Sleep(10);
            VectorAddition(v1, v2, add, size);
            VectorSubtraction(v1, v2, sub, size);
            VectorMultiplication(v1, v2, mult, size);
            clock_t end = clock();
            sequential_time = ((double)(end - start)) / CLOCKS_PER_SEC;

            // Parallel operations
            start = clock();Sleep(10);
            #pragma omp parallel sections
            {
                #pragma omp section
                VectorAddition(v1, v2, add, size);
                #pragma omp section
                VectorSubtraction(v1, v2, sub, size);
                #pragma omp section
                VectorMultiplication(v1, v2, mult, size);
            }
            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;
}
```

Vector Size	Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency
200	2	0.018000	0.024000	0.750000	0.375000
200	4	0.010000	0.011000	0.909091	0.227273
200	6	0.010000	0.012000	0.833333	0.138889
200	8	0.019000	0.021000	0.904762	0.113095
400	2	0.010000	0.016000	0.625000	0.312500
400	4	0.010000	0.010000	1.000000	0.250000
400	6	0.011000	0.012000	0.916667	0.152778
400	8	0.010000	0.019000	0.526316	0.065789
600	2	0.020000	0.010000	2.000000	1.000000
600	4	0.010000	0.013000	0.769231	0.192308
600	6	0.016000	0.021000	0.761905	0.126984
600	8	0.013000	0.016000	0.812500	0.101563
800	2	0.019000	0.014000	1.357143	0.678571
800	4	0.013000	0.013000	1.000000	0.250000
800	6	0.015000	0.012000	1.250000	0.208333
800	8	0.012000	0.010000	1.200000	0.150000
1000	2	0.013000	0.016000	0.812500	0.406250
1000	4	0.015000	0.012000	1.250000	0.312500
1000	6	0.012000	0.014000	0.857143	0.142857
1000	8	0.015000	0.011000	1.363636	0.170455

2) Write a parallel program using OpenMP to find sum of N numbers using the following constructs/clauses.

a. Critical section
b. Atomic
c. Reduction
d. Master
e. Locks

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include<windows.h>

int main ()
{
   int sum = 0;
   clock_t start, end;
    double cpu_time_used=0;
    int n;
    printf("Enter N: ");
    scanf("%d",&n);
   int total_threads_used;/*
   #pragma omp master
   {start = clock();Sleep(10);
    omp_lock_t writelock;
     omp_init_lock(&writelock);
   #pragma omp parallel num_threads(n) reduction (+:sum)
   {
        total_threads_used = n;
        #pragma omp for
        for(int i=1;i<=n;i++){
            omp_set_lock(&writelock);
            #pragma omp critical
            #pragma omp atomic
            sum = sum + i;
            omp_unset_lock(&writelock);
        }
    }
    omp_destroy_lock(&writelock);
    }*/
```

```
    start = clock();
    Sleep(10);
    #pragma omp for reduction(+:sum)
        for(int i=1;i<=n;i++){sum = sum + i;}
    printf("\nThe total sum is %d\n", sum);
    end = clock();
    cpu_time_used=cpu_time_used +((double) (end - start)) /
CLOCKS_PER_SEC;
    printf("\nTime taken: %0.3f\n",cpu_time_used);
    return 0;
}
```

```
Enter N: 50000

The total sum is 1250025000

Time taken: 0.017
```

```
Enter N: -3

The total sum is 0

Time taken: 0.012
```

3) Write a parallel program using OpenMP to implement the Odd-even transposition sort. Vary the input size and analyse the program efficiency.

```
#include<stdlib.h>
#include<stdio.h>
#include<time.h>
#include<string.h>
#include <omp.h>
#include<windows.h>

#define MAX_VALUE 1000

void odd_even_sort(int* a, int n)
{
    int phase, i, temp;
    for(phase = 0; phase < n; phase++)
    {
        if(phase % 2==0) //even phase
        {
            #pragma omp parallel for shared(a, n) private(i, temp)
```

```
            for(i = 1; i < n; i += 2)
            {
                if(a[i-1] > a[i])
                {
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
            }
        }
        else //odd phase
        {
            #pragma omp parallel for shared(a, n) private(i, temp)
            for(i = 1; i < n-1; i += 2)
            {
                if(a[i] > a[i+1])
                {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
            }
        }
    }
}

void generate_array(int* a, int size)
{
    int i = 0;
    srand(time(NULL));
    for(i = 0; i < size; i++)
    {
        a[i] = rand() % MAX_VALUE;
    }
}

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

```
    for (int size = 200; size <= 800; size += 200) {
        for (int threads = 2; threads <= size; threads *= 2) {
            double sequential_time = 0, parallel_time = 0;
            int *a = (int*)calloc(size, sizeof(int));
            int *initial = (int*)calloc(size, sizeof(int));

            generate_array(a, size);
            memcpy(initial, a, size * sizeof(int));

            // Sequential operation
            clock_t start = clock();Sleep(10);
            odd_even_sort(a, size);
            clock_t end = clock();
            sequential_time = ((double)(end - start)) / CLOCKS_PER_SEC;

            // Parallel operation
            #pragma omp parallel
            start = clock();Sleep(10);
            odd_even_sort(a, size);
            end = clock();
            parallel_time = ((double)(end - start)) / CLOCKS_PER_SEC;

            double speedup = sequential_time / parallel_time;
            double efficiency = speedup / threads;

            printf("%d\t\t%d\t%.6f\t\t%.6f\t\t%.6f\t%.6f\n", size,
threads, sequential_time, parallel_time, speedup, efficiency);

            free(a);
            free(initial);
        }
    }

    return 0;
}
```

Array Size	Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency
200	2	0.012000	0.013000	0.923077	0.461538
200	4	0.013000	0.013000	1.000000	0.250000
200	6	0.012000	0.013000	0.923077	0.153846
200	8	0.010000	0.013000	0.769231	0.096154
400	2	0.013000	0.012000	1.083333	0.541667
400	4	0.016000	0.014000	1.142857	0.285714
400	6	0.013000	0.012000	1.083333	0.180556
400	8	0.010000	0.016000	0.625000	0.078125
600	2	0.013000	0.012000	1.083333	0.541667
600	4	0.010000	0.013000	0.769231	0.192308
600	6	0.013000	0.014000	0.928571	0.154762
600	8	0.018000	0.014000	1.285714	0.160714
800	2	0.013000	0.013000	1.000000	0.500000
800	4	0.013000	0.012000	1.083333	0.270833
800	6	0.014000	0.012000	1.166667	0.194444
800	8	0.013000	0.012000	1.083333	0.135417
1000	2	0.014000	0.012000	1.166667	0.583333
1000	4	0.019000	0.021000	0.904762	0.226190
1000	6	0.014000	0.012000	1.166667	0.194444
1000	8	0.014000	0.012000	1.166667	0.145833

4) Write an OpenMP program to find the Summation of integers from a given interval. Analyze the performance of various iteration scheduling strategies.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include<windows.h>
```

```c
int main ()
{
   int sum = 0;
   clock_t start, end;
    double cpu_time_used=0;
    int low,high;
    printf("Enter Lower Limit: ");
    scanf("%d",&low);
    printf("Enter Uppper Limit: ");
    scanf("%d",&high);
   int total_threads_used;
    start = clock();
    Sleep(10);
   #pragma omp parallel num_threads(high-low+1) reduction (+:sum)
   {
        total_threads_used = high-low+1;
        #pragma omp for
        for(int i=low;i<=high;i++){
            sum = sum + i;
        }
    }
    printf("\n\nThe total sum is %d\n\n\n", sum);
    end = clock();
    cpu_time_used=cpu_time_used +((double) (end - start)) /
CLOCKS_PER_SEC;
    printf("\nTime taken: %0.3f\n",cpu_time_used);
    return 0;
}
```

```
Enter Lower Limit: 300
Enter Uppper Limit: 5000


The total sum is 12457650



Time taken: 0.013
```

```
Enter Lower Limit: -100
Enter Uppper Limit: 100


The total sum is 0



Time taken: 0.011
```

5) Write a parallel program using OpenMP to generate the histogram of the given array A.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>

#define MAX_VALUE 10

void generate_array(int* a, int size)
{
    int i = 0;
    srand(time(0));
    for(i = 0; i < size; i++)
    {
        a[i] = rand() % MAX_VALUE;
    }
}

int main() {
    printf("Array Size\tThreads\tTime (s)\n");
    int temp;
    for (int size = 200; size <= 1200; size += 200) {
        for (int num_threads = 2; num_threads <= 8; num_threads += 2) {
            double cpu_time_used = 0;
            int *array = (int*)calloc(size, sizeof(int));
            generate_array(array, size);

            clock_t start = clock();Sleep(10);
            #pragma omp parallel for num_threads(num_threads)
shared(array, size) private(i, j, size, temp)
            for (int i = 0; i < size; ++i) {
                for (int j = 0; j < size; ++j) {
                    if (array[j] < temp) {
                        // Do nothing
                    } else {
                        // Print #
                        --array[j];
                    }
```

```
                    --temp;
                }
            }
            clock_t end = clock();
            cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
            printf("%d\t\t%d\t%.6f\n",  size, num_threads, cpu_time_used);

            free(array);
        }
    }
    return 0;
}
```

Array Size	Threads	Time (s)
200	2	0.010000
200	4	0.012000
200	6	0.013000
200	8	0.013000
400	2	0.014000
400	4	0.013000
400	6	0.013000
400	8	0.017000
600	2	0.013000
600	4	0.011000
600	6	0.012000
600	8	0.014000
800	2	0.014000
800	4	0.013000
800	6	0.014000
800	8	0.013000
1000	2	0.013000
1000	4	0.013000
1000	6	0.015000
1000	8	0.014000
1200	2	0.014000
1200	4	0.013000
1200	6	0.014000
1200	8	0.013000